Red Flag Server 4.0

高级管理手册

北京中科红旗软件技术有限公司编写

地址：中国北京海淀区万泉河路 68 号紫金大厦 6 层

Red Flag Software Co., Ltd.

http://www.redflag-linux.com

声明：

目　录

序

欢迎使用 Red Flag Server 4.0 操作系统！

《Red Flag Server 4.0 高级管理手册》描述了 Red Flag Advanced Server 4.0 操作系统的高级特性，提供了定制这些高级特性的相关背景知识和操作步骤说明。主要内容包括：

- 高级特性概述
- 高级文件系统指南
- 创建软件 RAID 设备
- LVM（逻辑卷管理）使用手册
- EVMS（企业卷管理系统）指南
- 使用内核升级工具
- 异步 I/O 指南

请注意：除 RAID 和 LVM 之外，本手册涉及的其他高级特征和管理工具都需要安装 Red Flag Advanced Server 4.0 的第二张光盘后才能够使用。

本书的适用对象

本手册讨论 Red Flag Server 4.0 系统的高级管理话题，适用于希望学习和使用系统高级特性的系统管理员和高级用户。

在阅读本手册之前，您应该已经掌握了 Red Flag Server 4.0 系统的基础操作。《Red Flag Server 4.0 系统管理手册》中涉及的系统管理基础知识及相关系统管理主题对理解本手册的内容很有帮助，建议您先阅读它。

印刷惯例

《Red Flag Server 4.0 高级管理手册》用不同的字体、大小和风格代表文件名、命令、菜单项和其它特殊元素，具体如下：

格式	含义	示例
command、filename、output message	系统命令、文件名或目录名、计算机的屏幕输出信息。	使用 ls -a 命令来查看当前工作目录中的所有文件； 编辑文件 /etc/fstab； [root@localhost /root]#
application	表示一个应用程序或实用工具的名称。	使用 **Kedit** 编辑文本文件。
<**key**> <**key1+key2**>	表示键盘上的按键和组合按键。	使用 <**Tab**> 键进行命令补全； 按 <**Ctrl+Shift**> 切换输入法类型。
"**Menu Item**"	界面上引用的文本、按钮和菜单项。	确认后按"**下一步**"继续。
→	连续菜单选择之间的分隔符。	"**新建→用户**"表示打开"**新建**"菜单，选择其中的"**用户**"子菜单项。
user input	用户在命令行或文本框中输入的内容。	在 boot：提示下键入 **expert** 命令，进入专家安装方式。

提示与警告

为了强调《Red Flag Server 4.0 高级管理手册》中的某些重要的信息，我们使用下面两种方式加以重点说明：



一些有用的额外信息、使用中的提示和帮助用户更加顺利完成工作的小技巧等。



看到这一标记时应特别注意，它表示一些重要的警告和错误提示信息。

第 1 章　高级特性概述

Red Flag Advanced Server 4.0 是 Red Flag Server 4.0 系列的核心产品，通过对核心的改进和优化获得了性能、可靠性和功能扩展性等方面的较大提升，并加入了一些有用的新特征。

本章概括地介绍了 Red Flag Advanced Server 4.0 系统的高级特性。

1.1　性能方面

1.1.1　 I/O 子系统优化

- **支持异步 I/O**

包括裸设备异步 I/O 和文件系统异步 I/O。使得数据库，文件服务器等应用可以利用异步 I/O，从而使得多个 I/O 操作同时执行，提高了系统的性能。

异步 I/O 允许用户在一个 I/O 操作处于运行状态时，执行另一个有用工作。也就是说，在异步 I/O 中，请求发送给操作系统，同时下一条指令立即执行。因为 I/O 操作和应用程序进程是同时进行的，所以使用异步 I/O 会提升性能，而且通常会提高系统的 I/O 吞吐量。

关于异步 I/O 的更多信息，请参阅本手册的第 7 章：异步 I/O 指南。

- **Fine grain I/O spinlock**

Linux 使用 spinlocks 保证内核数据结构的完整性。在 linux 系统运行中，多个进程要共享内核的一些资源，如块设备、网络设备等。为保证多个进程的正确运行，linux 采用了 spinlock 来保护这些共享资源：当一个进程获取对某一资源的使用权后，在对资源操作前，首先对该资源加锁，防止其它进程对该资源的访问；在完成操作后，释放资源然后解锁，使其它进程可以访问该资源。在某一资源被锁定的过程中，只有加锁的进程可以访问它，其它进程必须等待，直到资源解锁才可由另一进程以同样的方式访问它。

在目前的 Linux 内核中，仅使用了一个 spinlock 来保护所有的块设备子系统，这将导致所有进行 I/O 操作的进程都要争用这个单一资源，即便它们使用互不相关的不同设备，从而使之成为影响系统 I/O 性能的瓶颈。

通过实现 fine-grained locking 机制：在块设备子系统中，不同的设备使用不同的 spinlock，使得在不同块设备上的 I/O 操作可以并行进行，从而消除了上面所述的瓶颈。

在 linux 内核中使用 fine-grain I/O splinlock 提高了系统 I/O 的整体性能，特别是在具有多个 I/O 控制器的 SMP 服务器上，可以显著提高系统 I/O 的带宽。

- **Bounce Buffer elimination**

在 IA-32 系统中，由低端开始的 1GB 物理内存称为 Low memory，1GB 以上的内存称为 high memory。Linux 早期版本中要求访问存储设备的数据缓冲区必须在物理内存的低端内存（low memory）中，即使应用程序可以同时使用 low memory 与 high memory。当数据缓冲区在 low memory 中时，访问它的所有的 I/O 请求产生直接的内存访问操作（no copy）。当应用程序的数据缓冲区在 high memory 中时，对它的 I/O 请求会导致内核在 low memory 中申请一块临时缓冲区，把数据从 high memory 中的缓冲区中复制到临时缓冲中才能写入块设备，或者从块设备中读出数据到临时缓冲区再复制到程序的缓冲区中。与在 low memory 的情况相比，这里增加了一个数据 copy 过程，称为 “bounce buffer”。bounce buffer 降低了程序的 I/O 性能，因为系统要不断在 low memory 中申请或释放大量的临时缓冲区，使 low memory 资源紧张，同时大量的数据复制加重了系统负载并降低了程序的性能。

bounce buffer elimination 使得在 high memory 中的数据缓冲区也可以直接与块设备相连，而不需要在 low memory 中的临时缓冲区。这可以降低系统的负载，提高应用程序的性能。

1.1.2　虚拟内存子系统优化

虚拟内存（VM）子系统在性能方面是十分关键的。例如，Oracle 数据库强烈依赖的 SGA（系统全局区域）的上限以及 OS 内存页大小等参数都是由虚拟内存子系统决定的。

- **在 4GB—64GB 内存系统上的超大容量内存管理（VLM）**

Red Flag Advanced Server 4.0 系统通过分配 SGA 和 shm，使得大型应用程序能够使用超过 4GB 的超大容量内存。在 VLM 下运行需要对系统本身进行一些设置更改；另外，还可能需要对应用程序的某些特性和参数做出限制。

对操作系统本身的修改：root 用户必需在 /dev/shm 上加载一个等于或大于共享内存数量的 in-memory 文件系统用于数据缓存。如果 in-memory 文件系统已经被加载，只要它足够大就可以一直使用。如果没有这样的文件系统，类似于下面的一个 mount 命令将工作：

```
mount -t shm shmfs -o nr_blocks=2097152 /dev/shm
```

该命令会在 /dev/shm 上创建一个 8GB 的 shmfs 文件系统，每个 nr_block 为 4096 bytes。

- **大页面**

在 Linux 核心中常规的页面尺寸为 4KB。Red Flag Advanced Server 4.0 支持 2MB 或 4MB 的大页面，对一些高要求的应用程序提高了内存应用效率。

1、 对于 4KB 页面的内存，IA-32 处理器使用 2-或 3-级的页表来完成虚拟地址向物理地址的映射。内含的 TLB（Translation Look-aside Buffer）入口技术可以完成虚拟地址转换，从而 CPU 将不需要到每个内存区去寻找物理地址。处理器中的 TLB 入口非常少，因此类似 Oracle 这样需要大量内存存取的应用调用 TLB 失败率很高。有了大页面支持，处理器就能够处理 4MB 内存

（或 2MB，和所选的模式有关）。这种情况下，TLB 中的 PTE （Page Table Entry）入口将可以覆盖到 4MB 而不是 4KB，这样，把地址 A，A+4KB，A+8KB 提高到了 A+4MB，TLB 错失率就会极大程度的降低。

2、 页表空间的大量减少提高了内存利用。这是因为 1024 个 4KB 的 PTE 现在由一个 4MB 的 PTE 来处理。

3、 大页面不会被交换出去，这意味着数据库的数据缓冲区（db_block_buffers）始终被锁在物理内存中，带来了更好的性能。同时，由于这些内存页不分配交换空间，意味着更多的可用交换空间和较少的页面缓存。

➢ 如何实现大页面：

1) 修改 /etc/lilo.conf 文件，在其中添加 append="bigpages=<size>M” 参数行，执行 lilo。
2) 在 /etc/sysctl.conf/ 中，添加 kernel.shm-use-bigpages = 1 或 2（1：shm 用 sysV 共享内存情况，2：shm 用 shmfs 情况；）。
3) 重启主机，检查 bigpages 和 shm-use-bigpages 已经改正。

➢ 关闭 bigpages 过程：

1) 修改 /etc/lilo.conf，去掉 append= “bigpages=<size>M” 参数行，执行 lilo。
2) 在 /etc/sysctl.conf/ 中，添加 kernel.shm-use-bigpages = 0。
3) 重启主机，检查 bigpages 和 shm-use-bigpages 已经改正。

1.1.3　进程调度优化

- **针对 SMP 优化的进程调度算法**

新的调度程序消除了大的进程队列锁，实现了每个 CPU 的进程队列和锁机制。这就允许并行的对每个 CPU 进行任务调度，而不会出现内部加锁（interlocking）现象，从而提高了 SMP 的可扩展性。

- **CPU affinity**

通常情况下，系统中的多个进程是以时间片为单位按照轮转的调度算法运行的。当一个进程运行时，系统会把它的数据与指令调入 CPU 的高速缓存中才能执行。在多 CPU 的 SMP 计算机上，一个进程在不同的运行时间片中可能会运行在不同的 CPU 上：当进程在下一时间片内仍在原 CPU 上时，该 CPU 高速缓存中的进程数据与指令可以立即使用，进程可以迅速继续运行；当进程在下一时间片中在不同的 CPU 上时，进程的指令与数据就需要重新装入新 CPU 的高速缓存中才可继续运行，进程的性能较低。这种反复的装入过程降低了程序的运行效率，也降低了 CPU 高速缓存的使用效率。

在 SMP 计算机上，通过为进程或线程指定运行的特定 CPU，可以避免进程在不同的时间片在不同的 CPU 上运行，提高 CPU 高速缓存的命中率减少进程调度争夺资源，提高应用程序的性能与系统吞吐量。

Linux 进程调度工具——schedutils 通过系统调用实现与进程调度相关的参数设置，如 CPU affinity。Schedutils 软件包中包括的程序有 taskset、irqset、lsrt 和 chrt，它们可以对处理进程参数进行全面的管理。

1、 taskset

taskset [options] [mask] [pid | command [arg]...]

设置/获取指定进程的 CPU 绑定或根据指定的绑定掩码运行一个新进程。

Mask：进程与 CPU 的绑定关系由一个 4 字节的无符号整数通过位掩码指定，每个二进制位代表一个逻辑处理器，最低端位指第一个逻辑处理器，最高端指最后一个处理器。如，当用十六进制表示时：

0x00000001　　指处理器 #0

0x00000003　　指处理器 #0 与 #1

0xFFFFFFFF　　指所有处理器（#0 到 #31）

选项：

-p，--pid　　针对指定 PID 的已有进程操作，而不是启动新任务。

-h，--help　　显示帮助信息。

-v，--version　　显示版本信息。

例子：

指定绑定 mask 运行新命令：

taskset [mask] -- [command] [arguments]

获取指定进程的绑定关系：

taskset -p [pid]

为已有进程设置 CPU 绑定关系：

taskset -p [mask] [pid]

2、 irqset

irqset [options] [mask] [interrupt]

设置/获取系统中断的 CPU 绑定。

通常在 SMP 系统中，系统中断完全是随机地被发送给任一处理器；而通过本工具可以把一个中断绑定到一组处理器，只有这些处理器可以接收并处理该中断。此处 mask 与 taskset 中意义相同。无参数运行 irqset 将显示所有中断的绑定关系。

1.2　功能方面

1.2.1　对文件系统的支持

支持多种最新的日志文件系统，包括 XFS、JFS、REISERFS、EXT3 等。关于这些日志文件系统的特征及比较，请参阅本手册第 2 章：高级文件系统指南。

1.2.2　支持 iSCSI 协议

对海量存储设备和相关协议有进一步的支持，包括 iSCSI。

iSCSI 允许在使用 TCP/IP 协议的网络上传输 SCSI I/O 命令和数据，就像将 SCSI 命令映射到光纤通道、并行 SCSI 和 SSA 介质上一样。在服务器端安装 iSCSI 设备驱动器，接收应用程序的 I/O 请求，再使用 iSCSI 协议将它们在 LAN 上传输。目标存储设备可以直接附着于 LAN，也可以使用路由器（协议，转换器）来连接到 LAN。与其它解决方案相比，iSCSI 有投资小、IO 性能突出、传输距离长、管理和部署方便等优点。

Red Flag Advanced Server 4.0 支持 iSCSI 的客户端，不支持服务端。

1.2.3　支持 LVM 和 EVMS

LVM（逻辑卷管理）大大增强了磁盘子系统的可伸缩性和管理效率，关于 LVM 的使用及其相关信息，请参阅本手册第 4 章：LVM 使用手册。

EVMS 是 IBM 开发的企业卷管理系统，它把各方面的卷管理技术，如磁盘分区、Linux 逻辑卷管理（LVM）、multi-disk（MD）管理、OS2 和 AIX 卷管理和文件系统操作统一在单个的包中。关于 EVMS 的使用及其相关信息，请参阅本手册第 5 章：使用 EVMS。

1.3　可靠性提升

采用了高内存 PTE 补丁，防止核心在负荷过大的情况下发生宕机。

在旧的内核中，Linux 只能在低端内存中分配页表入口（Page Table Entries，PTEs），这就有一个 1GB 的限制。在类似 Oracle 9iR2 等应用中，要使用大量的内存和进程，PTE 的总空间很大；同时连接数据库的用户量很大，核心运行时将可能超出 PTE 的空间，即使有可用的空闲内存和交换空间系统还是会

被挂起或宕掉。

高内存 PTE 补丁允许 VM 为分配 PTE 而使用 “高端内存” 共享池。当越来越多的用户连接到数据库并产出了额外的进程时，储存 PTE 的区域就溢出到高端内存，这样就允许系统比使用旧的内核多支持 3 到 5 倍数据的用户。



请阅读 《Red Flag Advanced Server 4.0 发行概述》，该文档中包括了 Red Flag Advanced Server 4.0 系统在产品定位、新特征和相关须知等方面的全面信息。

第 2 章　高级文件系统指南

Redflag Advanced Server 4.0 支持多种最新的日志文件系统，包括 XFS、JFS、REISERFS、EXT3 等。本章向您介绍这些日志文件系统的特性及简单的使用方法，以便尽可能轻松愉快地使用最新的文件系统技术。

在此之前，先介绍一些必要的基本知识，以帮助更好的理解。

2.1　日志系统（Journaling）

日志是一项非常重要的技术，在 ReiserFS、XFS、JFS、ext3 等文件系统中都用到它，用以达到快速检查文件系统一致性的目的。

2.1.1　元数据（Meta-data）

文件系统的存在允许用户储存、检索和操作数据，为此文件系统需要保持一个内在的数据结构使得数据有组织并且便于访问。这一内部的数据结构（确切地说就是 “**关于数据的数据**”）被称为元数据，它为文件系统提供了其特定的身份和性能特征。

通常，我们并不直接和文件系统的元数据打交道。而是一个特别的 Linux 文件系统驱动程序为我们完成相应的工作。Linux 文件系统驱动程序是专门用来操作复杂的元数据的。然而，为了使得文件系统驱动程序正常工作，有一个很重要的必要条件；它需要在某种合理的、一致的和没有干扰的状态下找到元数据。否则，文件系统驱动程序就不能理解和操作元数据，那么也就不能存取文件了。

2.1.2　fsck

当 linux 系统关闭时，内核驱动会把所有的缓冲区数据转送到磁盘，并确保文件系统被彻底卸载，这样文件系统的元数据将会处于可用的状态，并可以在下次启动时被正常装载及使用。但当一些意外发生情况发生时，文件系统没有被彻底卸载，元数据可能是错误的，这时就需要用 fsck 对文件系统进行全面地检查，修正找到的任何错误，使元数据恢复一致。

fsck 的工作就是确保要装载的文件系统的元数据是处于可使用的状态。典型方式是，fsck 扫描那些将被装载的文件系统，确定它们已被彻底卸载，并做出合理的假设——所有的元数据都没有问题；当 fsck 检测到没有被彻底卸载的文件系统时，就会彻底的扫描并且全面地检查该文件系统的元数据，修正这一过程中找到的任何错误；一旦 fsck 完成工作，文件系统就可以使用了。

尽管意想不到的电源故障或者系统挂起可能造成最近修改的数据丢失，但是由于元数据现在是一致的，文件系统可以被装载和使用。

使用 fsck 可以确保文件系统的一致性，但却不是最佳的解决方案。使用 fsck 所面临的问题是：fsck 必须扫描文件系统的全部元数据，才能确保文件系统的一致性。

对文件系统所有的元数据做彻底的一致性检查是一项极为费时的工作，文件系统越大，完成扫描所

花费的时间就越长。当 fsck 运行时，系统实际上是被中断了而不能工作，如果有一个庞大的文件系统，可能就会花上半个小时或者更长的时间来执行 fsck，这通常是难以接受的。

2.1.3　日志（Journal）

日志文件系统通过增加一个叫做日志的新的数据结构来解决 fsck 问题。这个日志是位于磁盘上的结构。在对元数据做任何改变以前，文件系统驱动程序会向日志中写入一个条目，这个条目描述了它将要做些什么。然后，它继续并修改元数据。通过这种方法，日志文件系统就拥有了近期元数据被修改的历史记录，当检查到没有彻底卸载的文件系统的一致性问题时，这个记录就大有用处了。

可以这样来看待日志文件系统——除了存储数据和元数据以外，它们还有一个日志——可以称之为元元数据。

fsck 如何处理日志文件系统呢？实际上，通常它什么都不做，只是忽略文件系统并允许它被装载。在快速地恢复文件系统到达一致性状态的背后，真正起作用的在于 Linux 文件系统驱动程序中。

当文件系统被装载时，Linux 文件系统驱动程序查看文件系统是否完好。如果由于某些原因出了问题，那么就需要对元数据进行修复，但不是执行对元数据的彻底扫描（就像 fsck 那样），而是查看日志。由于日志中包含了按时间顺序排列的近期的元数据修改记录，它就简单地查看最近被修改的那部分元数据。因而，它能够在几秒钟时间内将文件系统恢复到一致性状态。并且与 fsck 所采用的传统方法不同，这个日志重放过程在大型的文件系统上并不需要花更多的时间。有了日志，数百 G 的文件系统元数据几乎能在瞬间恢复到一致性的状态。

2.1.4　不同的日志文件系统

哪种 Linux 日志记录文件系统是“最好的”？没有一个对每个应用程序都“合适的”文件系统。每个文件系统都有自身的长处，所以，理解每种文件系统的长处和弱点，以便对使用哪种文件系统做出一个有根据的选择，远远优于选出一个绝对的“最好的”文件系统，并将它用于所有可能的应用程序。

2.2　ReiserFS

ReiserFS 文件系统堪称最有魄力的日志文件系统开发项目，因为它不只是将现有的日志文件系统（如：XFS、JFS）移植到 Linux 内核，它的设计也不象 ext3 那样基于早先的文件系统。相反，ReiserFS 的设计完全是从头开始的。

ReiserFS 3.6.x（作为 Linux 2.4 一部分的版本）是由 Hans Reiser 和他的在 Namesys 的开发组共同开发设计的。Hans 和他的组员们相信最好的文件系统是那些能够有助于创建独立的共享环境或者命名空间的文件系统，应用程序可以在其中更直接、有效和有力地相互作用。为了实现这一目标，文件系统就应该满足其使用者对性能和功能方面的需要。那样，使用者就能够继续直接地使用文件系统，而不必建造运行在文件系统之上（如数据库之类）的特殊目的层。

2.2.1　Reisefs 的特点

通常，像 ext2 和 ufs 这样的文件系统在处理小文件这一方面做的并不是很好。ext2 擅长存储大量大小在 20k 以上的文件，但是对于存储类如 2,000 个 50 字节的小文件这种情况来说，它就不是很理想

了。当 ext2 处理非常小的文件时，不仅性能显著地下降，存储效率也同样下降，因为 ext2 是按 1k 或者 4k 的块为单位来分配空间的（在文件系统创建时设定）。这迫使开发人员转向数据库或者特别组织的处理来获取他们所需要的某种性能，这种 “围绕问题进行编码” 的方法促使了代码的膨胀，并产生了许多不兼容的特殊 API。

ReiserFS 几乎在各个方面都优于 ext2，但是在处理小文件（4K 以下）时才真正体现出了闪光点。在实际运用中，ReiserFS 的小文件性能好得让人吃惊。当处理小于 1k 的文件时，ReiserFS 大概要比 ext2 快 8 到 15 倍！而且这些性能提高并不以其它文件类型的性能损失为代价。

ReiserFS 的弱点：在当前的 ReiserFS（版本 3.6）中，与 ext2 和 ext3 相比，尤其是读取大的邮件目录时，特定文件访问模式有可能导致特别糟糕的性能；ReiserFS 没有好的 NFS 兼容性跟踪记录；稀疏文件性能也较差。

2.2.2　ReiserFS 技术

ReiserFS 使用了特殊的优化 b* 平衡树（每个文件系统一个）来组织所有的文件系统数据，这为其自身提供了非常不错的性能改进，也减轻了文件系统设计上的人为约束。例如，现在一个目录下可以容纳 100,000 个子目录。另一个使用 b* 树的好处就是 ReiserFS 能够像大多数其它的下一代文件系统一样，根据需要动态地分配索引节，而不必在文件系统创建时建立固定的索引节。这有助于文件系统更灵活地适应其面临的各种存储需要，同时提供附加的空间有效率。ReiserFS 里的目录是完全动态分配的，因此不存在 ext2 中常见的无法回收巨型目录占用的磁盘空间的情况。

ReiserFS 有许多特征是特别针对提高小文件的性能的。和 ext2 不同，ReiserFS 并不固定地以 1k 或者 4k 的块分配存储空间，而是分配所需要的精确尺寸。而且 ReiserFS 也包括了以尾文件为中心的特殊优化——尾文件是指那些比文件系统块小的文件及文件结尾部分。为了提高性能，ReiserFS 能够在 b* 树的叶子节点存储文件，而不是把数据存储在磁盘的其它地方再指向它。

这样做有两个优点。第一，它显著地提高了小文件的性能。由于文件数据和 stat_data（索引节）信息是紧挨着存储的，它们通常能被同一次磁盘 IO 操作所读取。第二，ReiserFS 能够压缩尾文件，节省大量磁盘空间。实际上，带有尾文件压缩功能（默认）的 ReiserFS 文件系统可以比同等的 ext2 文件系统多存储 6 个百分点的数据。

然而，由于在文件被修改时，尾文件压缩迫使 ReiserFS 重装数据，这就导致了性能上的轻微折损。为此，ReiserFS 尾文件压缩功能可以被关掉，从而允许系统管理员在速度与空间有效率上做出选择，或者牺牲一些存储能力来换取更高的速度。

据 HansReiser 和他的开发者小组所说，预计在 2.4.20_pre1 会有一些非常好的改进，包括对 Chris Mason 的数据日志（象 ext3 的 “data=journal” 模式）的支持、伸缩性要好得多的新的块分配代码以及对大文件性能方面的一些改进，预计从 IDE 驱动器读取大文件时性能提高要高达 15%。

2.2.3　ReiserFS 工具

对 reiseFS 的支持包括内核中的驱动与 reiserfs-utils 工具。在 linux 2.4.18 内核中已经有了一个非常稳定的 ReiserFS 实现，而在 reiserfs-utils 中包括的用户工具有：

- **mkreiserfs**

用于在指定的设备上创建 Reiserfs。直接使用 mkreiserfs 等价于 mkfs -t reiserfs。mkreiserfs 的命令行选项包括：

```
mkreiserfs [-dfV] [-b|--block-size N] [-h | --hash HASH] \
        [-u | --uuid UUID] [-l|--label LABEL] [--format FORMAT ] \
        [-j | --journal-device FILE] [-s | --journal-size N] \
        [-o | --journal-offset N] [-t | --transaction-max-size N] \
        device [filesystem-size]
```

其中各选项的含义请参见 mkreiserfs 命令的 man page。

- **reiserfsck**

可以直接使用 reiserfsck 或使用 fsck -t reiserfs 对一个 Reisefs 文件系统进行检查。reiserfsck 的命令行选项包括：

```
reiserfsck [-afprVy] \
        [--check | --fix-fixable | --rebuild-sb | --rebuild-tree |
        --clean-attributes] \
        [-j | --journal-device device] \
        [--no-journal-available] [-z | --adjust-file-size] \
        [-S | --scan-whole-partition] [-l | --logfile filename] \
        [-n | --nolog] [-q | -quiet] device
```

其中各选项的含义请参见 reiserfsck 命令的 man page。

- **resize_reiserfs**

改变指定 reiserfs 文件系统的大小。其命令行参数为：

```
resize_reiserfs [-s [+|-]size[K|M|G]] [-j dev] [-fqv] device
```

在使用 LVM 时，这个命令会使用到。

- **debugreiserfs**

一般在调试 reiserfs，解决文件系统的问题时使用。

2.2.4　使用简介

- **创建 Reiserfs 文件系统**

要使用新的文件系统，首先使用以下命令创建一个 Reiserfs 文件系统：

mkreiserfs -f /dev/sda3

在出现版本信息和 Reiserfs 参数后，将出现警告信息：

ATTENTION: YOU SHOULD REBOOT AFTER FDISK!

(y/n) ALL DATA WILL BE LOST ON '/dev/sda3'!

键入 “**y**” 后回车，开始创建新的文件系统。

格式化的过程比较慢，视分区的大小而定。

- **使用 reiserfs 文件系统**

为了使用新的分区，需要将其 mount 到系统中来：

mounut –t reiserfs /dev/sda3 /mnt/fs

这样新的文件系统就被挂载到 /mnt/fs 目录下了。现在就可以开始使用新的文件系统存放文件了。

2.3　XFS

XFS 最初是由 Silicon Graphics，Inc.（SGI）于 90 年代初开发的。那时，SGI 发现他们的现有文件系统（existing filesystem，EFS）正在迅速变得不适应当时激烈的计算竞争。为此，SGI 决定设计一种全新的高性能 64 位文件系统，而不是去调整 EFS 在先天设计上的某些缺陷。因此，XFS 诞生了，并于 1994 年随 IRIX 5.3 的发布而应用于计算。它至今仍作为 SGI 基于 IRIX 的产品的底层文件系统来使用。

现在，XFS 也可以用于 Linux。XFS 的 Linux 版的到来是激动人心的，首先因为它为 Linux 社区提供了一种健壮的、优秀的以及功能丰富的文件系统，并且这种文件系统所具有的可伸缩性能够满足最苛刻的存储需求。

2.3.1　XFS 设计

在 “Scalability in the XFS Filesystem” 一文中，SGI 工程师解释：他们设计 XFS 的主要思想只有一个，那就是：“考虑大东西”。XFS 的设计消除了传统文件系统中的一些限制。

2.3.1.1　分配组（allocation groups）

当创建 XFS 文件系统时，底层块设备被分割成八个或更多大小相等的线性区域（region）。可以将它们想象成“块”（chunk）或者“线性范围（range）”，但是在 XFS 中，每个区域称为一个“分配组”。

分配组是唯一的，因为每个分配组管理自己的索引节点（inode）和空闲空间，实际上，是将这些分配组转化为一种文件子系统，这些子系统正确地透明存在于 XFS 文件系统内。

2.3.1.2　分配组与可伸缩性

XFS 使用分配组，以便能有效地处理并行 IO。因为，每个分配组实际上是一个独立实体，所以内核可以同时与多个分配组交互。如果不使用分配组，XFS 文件系统代码可能成为一种性能瓶颈，迫使大量需求 IO 的进程“排队”来使索引节点进行修改或执行其它种类的元数据密集操作。多亏了分配组，XFS 代码将允许多个线程和进程持续以并行方式运行，即使它们中的许多线程和进程正在同一文件系统上执行大规模 IO 操作。因此，将 XFS 与某些高端硬件相结合，将获得高端性能而不会使文件系统成为瓶颈。分配组还有助于在多处理器系统上优化并行 IO 性能，因为可以同时有多个元数据更新处于“在传输中”。

2.3.1.3　无处不在的 B+树

分配组在内部使用高效的 B+ 树来跟踪主要数据，譬如空闲空间的范围和索引节点。实际上，每个分配组使用两棵 B+ 树来跟踪空闲空间：

- 一棵树按空闲空间的大小排序来存储空闲空间的范围；
- 另一棵树按块设备上起始物理位置的排序来存储这些区域。

XFS 擅长于迅速发现空闲空间区域，这种能力对于最大化写性能很关键。

对索引节点进行管理时，XFS 也是很有效的。每个分配组在需要时以 64 个索引节点为一组来分配它们。每个分配组通过使用 B+ 树来跟踪其索引节点，该 B+ 树记录着特定索引节点号在磁盘上的位置。我们会发现 XFS 之所以尽可能多地使用 B+ 树，原因在于 B+ 树的优越性能和极大的可扩展性。

2.3.1.4　日志记录

XFS 是一种日志记录文件系统，它允许意外重新引导后的快速恢复。象 ReiserFS 一样，XFS 使用逻辑日志；即，它不象 ext3 那样将文字文件系统块记录到日志，而是使用一种高效的磁盘格式来记录元数据的变动。就 XFS 而言，逻辑日志记录是很适合的；在高端硬件上，日志经常是整个文件系统中争用最多的资源。通过使用节省空间的逻辑日志记录，可以将对日志的争用降至最小。另外，XFS 允许将日志存储在另一个块设备上，例如，另一个磁盘上的一个分区。这个特性很有用，它进一步改进了 XFS 文件系统的性能。

象 ReiserFS 一样，XFS 只对元数据进行日志记录，并且在写元数据之前，XFS 不采取任何专门的预防措施来确保将数据保存到磁盘。这意味着，使用 XFS（就像使用 ReiserFS）时，如果发生意外的重新引导，则最近修改的数据有可能丢失。然而，XFS 日志有两个特性使得这个问题不象使用 ReiserFS 时那么常见。

使用 ReiserFS 时，意外重新引导可能导致最近修改的文件中包含先前删除文件的部分内容。除了数据丢失这个显而易见的问题以外，理论上，还可能引起安全威胁。相反，当 XFS 日志系统重新启动时，XFS 确保任何未写入的数据块在重新引导时置零。因此，丢失块由空字节来填充，从而消除了安全

漏洞。

那么，关于数据丢失问题本身，该怎么办呢？通常，使用 XFS 时，该问题被最小化了，原因在于：XFS 通常比 ReiserFS 更频繁地将暂挂元数据更新写到磁盘，尤其是在磁盘高频率活动期间。因此，如果发生死锁，那么，最近元数据修改的丢失，通常比使用 ReiserFS 时要少。当然，这不能彻底解决不及时写数据块的问题，但是，更频繁地写元数据也确实促进了更频繁地写数据。

对于 XFS 1.1，文件系统的元数据只在两种情况下会同步（有序）更新：

- 如果文件系统需要分配空间，并且有一个紧随的事务来释放同一块空间；
- 在 XFS 处理用 O_SYNC（同步）选项打开的文件事务的时候；在这种情况下，对这个文件进行写操作将会使得文件系统对元数据所做的其它任何紧随其后的更改被刷新到磁盘。

幸运的是，绝大多数典型的服务器的 I/O 操作在本质上都是异步的。

2.3.1.5　延迟分配

延迟分配是 XFS 独有的特性。分配（allocation）指：查找空闲空间区域并用于存储新数据的过程。

XFS 通过将分配过程分成两个步骤来处理。首先，当 XFS 接收到要写入的新数据时，会在 RAM 中记录暂挂事务，并只在底层文件系统上保留适当空间。然而，尽管 XFS 为新数据保留了空间，但它却没有决定将什么文件系统块用于存储数据，至少现在还没决定。XFS 进行拖延，将这个决定延迟到最后可能的时刻，即直到该数据真正写到磁盘之前做出。

通过延迟分配，XFS 赢得了许多机会来优化写性能。到了要将数据写到磁盘的时候，XFS 能够以这种优化文件系统性能的方式，智能地分配空闲空间。尤其是，如果要将一批新数据添加到单一文件，XFS 可以在磁盘上分配一个单一、相邻区域来储存这些数据。如果 XFS 没有延迟它的分配决定，那么，它也许已经将数据写到了多个非相邻块中，从而显著地降低了写性能。但是，因为 XFS 延迟了它的分配决定，所以，它能够一下子写完数据，从而提高了写性能，并减少了整个文件系统的碎片。

在性能上，延迟分配还有另一个优点。在要创建大量临时文件的情况下，XFS 可能根本不需要将这些文件全部写到磁盘。因为从未给这些文件分配任何块，所以，也就不必释放任何块，甚至根本没有触及底层文件系统元数据。

2.3.2　EA/ACL

XFS 的一个优点是它包含名为“访问控制表”或 ACL 的特殊功能，现在 XFS 文件系统上缺省启用这些功能。访问控制表允许定义细粒度的文件许可权。例如，现在您不再仅限于为用户、组及所有其它人定义“rwx”访问权限，而可以添加任意数目的额外用户或组并为它们指定“rwx”许可权。

XFS 包含另外一个名为“扩展属性”的优秀功能特性。扩展属性让您将用户定义的数据同文件系统对象相关联。例如，如果有一幅名为 mygraphic.png 的图像，可以附加一个名为“thumbnail”的属性，它包含该图像的一个小版本。普通文件 IO 操作将看不到这个数据，但任何程序都可以使用一个特殊的扩展属性 API 来访问它。扩展属性在某些方面类似 MacOS 系统中的“资源分支”。

2.3.3 限制

目前，XFS 的 Linux 版本要求 XFS 文件系统块的大小与平台的内存页面大小相同。这常常使得在 x86 系统上的磁盘不可能移到 Itanium 系统上，因为 Itanium 可以使用最大 64K 的页面，而 x86 只能用 4K。此外，对于大多数的任务来说，大小为 64K 的文件系统块并非最佳选择，但当前的代码迫使一些 Itanium 系统必须使用这样的文件系统块大小。如果修复了这个块大小问题，不仅将 XFS 文件系统从 x86 迁移到 ia64 上变得容易了，而且提供了一个额外的好处，就是允许系统管理员选择适合于他们的需要的 XFS 文件系统块大小。

2.3.4 XFS 工具

XFS 文件系统的工具包为 xfsprogs，它包括：

mkfs.xfs	在指定的块设备上创建 XFS 文件系统，也可通过 mkfs -t xfs 使用。
fsck.xfs	这个工具由 fsck -t xfs 调用，它什么也不做。
xfs_repair	检修 XFS 文件系统，可以修复已损坏的 XFS 文件系统。
xfs_admin、xfs_bmap、xfs_check、xfs_db、xfs_freeze、xfs_growfs、xfs_info、xfs_logprint、xfs_mkfile、xfs_ncheck、xfs_rtcp	这些工具可以改变 XFS 文件系统的参数，完成 XFS 文件系统的管理和调试工作。



请注意：在 Linux 系统中，目前 XFS 文件系统的块大小最大只支持到 4K，这也是创建文件系统时的默认值。

2.4 JFS

日志文件系统（JFS）最初是 IBM 的 AIX 所使用的文件系统，并在 AIX 上经过了长期使用，被证明是快速和可靠的。现在 IBM 把它移植到了 Linux 上。

如果发生系统崩溃，JFS 提供了快速文件系统重启。通过使用数据库日志技术，JFS 能在几秒或几分钟之内把文件系统恢复到一致状态，而非日志文件系统却要花上几小时甚至几天才能完成。

2.4.1 设计特性（Design features）

JFS 从一开始就设计成完全集成了日志记录，而不是在现有文件系统上添加日志记录。JFS 的许多特性使之区别于其它文件系统。

2.4.2 日志处理（Journaling）

JFS 提供了改进的结构化一致性和可恢复性，以及比非日志文件系统（如：HPFS、ext2 和传统 UNIX 文件系统）快得多的系统重启时间。发生系统故障时，非日志文件系统容易崩溃，是由于一个逻

辑写文件操作通常占用多个媒体 I/O 来完成，且在任何给定时间，可能没有完全反映在媒体上。这些文件系统依靠重启实用程序（也就是 fsck），fsck 检查文件系统的所有元数据（如：目录和磁盘寻址结构）以检测和修复结构完整性问题。这是一个耗时并且容易出错的过程，在最糟糕的情况下，它还可能丢失或放错数据。

相反，JFS 使用原来为数据库开发的技术，记录了文件系统元数据上执行的操作（即原子事务）信息。如果发生系统故障，可通过重放日志并对适当的事务应用日志记录，来使文件系统恢复到一致状态。由于重放实用程序只需检查文件系统最近活动所产生的运行记录，而不是检查所有文件系统的元数据，因此，与这种基于日志的方法相关的文件系统恢复时间要快得多。

基于日志恢复的其它几个方面也值得注意。首先，JFS 只记录元数据上的操作，因此，重放这些日志只能恢复文件系统中结构关系和资源分配状态的一致性。它没有记录文件数据，也没有将这些数据恢复到一致状态。因此，恢复后某些文件数据可能丢失或失效，对数据一致性有关键性需求的用户应该使用同步 I/O。

面对媒体出错，日志记录不是特别有效。特别地，在将日志或元数据写入磁盘的期间发生的 I/O 错误，意味着在系统崩溃后，要将文件系统恢复到一致状态，需要耗时并且有可能强加的全面完整性检查。这暗示着，坏块重定位是任何驻留在 JFS 下的存储管理器或设备的一个关键特性。

JFS 日志记录的语义如下：当涉及元数据更改的文件系统操作，例如，unlink() 返回成功执行的返回值时，操作的结果已经提交到文件系统，即使系统崩溃了也可以发现。例如，一旦成功删除了文件，即使系统崩溃然后重启，它仍然是删除的并且不会再重新出现。

日志记录风格将同步写入日志磁盘引入每个修改元数据的 inode 或 vfs 操作。在性能方面，与依赖（多个）谨慎的同步元数据写操作以获得一致性的许多非日志文件系统相比，这种方法较好。但是，与其它日志文件系统相比，它在性能上处于劣势。其它日志文件系统，如 Veritas VxFS 和 Transarc Episode，使用不同的日志风格并且缓慢地将日志数据写入磁盘。

在执行多个并行操作的服务器环境中，通过将多个同步写操作组合成单一写操作的组提交来减少这种性能损失。JFS 日志记录风格随着时间推移而得到不断改进，现在提供了异步日志记录，异步日志记录提高了文件系统的性能。

2.4.2.1　基于盘区的寻址结构（Extent-based addressing structures）

JFS 使用基于盘区的寻址结构，连同主动的块分配策略，产生紧凑、高效、可伸缩的结构，以将文件中的逻辑偏移量映射成磁盘上的物理地址。盘区是象一个单元那样分配给文件的相连块序列，可用一个由 ＜逻辑偏移量，长度，物理地址＞ 组成的三元组来描述。寻址结构是一棵 B+ 树，该树由盘区描述符（上面提到的三元组）填充，根在 inode 中，键为文件中的逻辑偏移量。

2.4.2.2　可变的块尺寸（Variable block size）

按文件系统分，JFS 支持 512、1024、2048 和 4096 字节的块尺寸，以允许用户根据应用环境优化空间利用率。较小的块尺寸减少了文件和目录中内部存储碎片的数量，空间利用率更高。但是，小块可能会增加路径长度，与使用大的块尺寸相比，小块的块分配活动可能更频繁发生。因为服务器系统通常主要考虑的是性能，而不是空间利用率，所以缺省块尺寸为 4096 字节。

2.4.2.3　动态磁盘 inode 分配（Dynamic disk inode allocation）

JFS 按需为磁盘 inode 动态地分配空间，同时释放不再需要的空间。这一支持避开了在文件系统创建期间，为磁盘 inode 留固定数量空间的传统方法，因此用户不再需要估计文件系统包含的文件和目录最大数目。另外，这一支持使磁盘 inode 与固定磁盘位置分离。

2.4.2.4　目录组织（Directory organization）

JFS 提供两种不同的目录组织。第一种组织用于小目录，并且在目录的 inode 内存储目录内容。这就不再需要不同的目录块 I/O，同时也不再需要分配不同的存储器。最多可有 8 个项可直接存储在 inode 中，这些项不包括自己（.）和父（..）目录项，这两个项存储在 inode 中不同的区域内。

第二种组织用于较大的目录，用按名字键控的 B+ 树表示每个目录。与传统无序的目录组织比较，它提供更快的目录查找、插入和删除能力。

2.4.2.5　稀疏和密集文件（Sparse and dense files）

按文件系统分，JFS 既支持稀疏文件也支持密集文件。

稀疏文件允许把数据写到一个文件的任意位置，而不要将以前未写的中间文件块实例化。所报告的文件大小是已经写入的最高块位处，但是，在文件中任何给定块的实际分配，只有在该块进行写操作时才发生。例如，假设在一个指定为稀疏文件的文件系统中创建一个新文件。应用程序将数据块写到文件中第 100 块。尽管磁盘空间只分配了 1 块给它，JFS 将报告该文件的大小为 100 块。如果应用程序下一步读取文件的第 50 块，JFS 将返回填充了 0 的一个字节块。假设应用程序然后将一块数据写到该文件的第 50 块，JFS 仍然报告文件的大小为 100 块，而现在已经为它分配了两块磁盘空间。稀疏文件适合需要大的逻辑空间但只使用这个空间的一个（少量）子集的应用程序。

对于密集文件，将分配相当于文件大小的磁盘资源。在上例中，第一个写操作（将一块数据写到文件的第 100 块）将导致把 100 个块的磁盘空间分配给该文件。在任何已经隐式写入的块上进行读操作，JFS 将返回填充了 0 的字节块，正如稀疏文件的情况一样。

2.4.3　JFS 内部/潜在限制（Internal /potential JFS limits）

JFS 是完全 64 位的文件系统。所有 JFS 文件系统结构化字段都是 64 位大小。这允许 JFS 同时支持大文件和大分区。

2.4.3.1　文件系统大小（file system size）

JFS 支持的最小文件系统是 16M 字节。最大文件系统的大小是文件系统块尺寸和文件系统元数据结构支持的最大块数两者的乘积。JFS 将支持最大文件长度是 512 万亿字节（TB）（块尺寸是 512 字节）到 4 千万亿字节（PB）（块尺寸是 4K 字节）。

2.4.3.2　文件长度（file size）

最大文件长度是主机支持的虚拟文件系统最大文件长度。例如：如果主机只支持 32 位，则这就限制了文件长度。

2.4.3.3　可移动介质（removable media）

JFS 不支持把软盘作为基本文件系统设备。

2.4.4　标准管理实用程序（Standard administrative utilities）

JFS 提供创建和维护文件系统的标准管理实用程序。

2.4.4.1　创建文件系统（create a file system）

这个实用程序提供 mkfs 命令的 JFS 特定部分，用来在指定的驱动器上初始化 JFS 文件系统。该实用程序在较低级别上操作，并假设文件系统所存在的任何卷的创建/初始化由更高级别的另一个实用程序处理。

2.4.4.2　检查/修复文件系统（check/recoverya file system）

这个实用程序提供 fsck 命令的 JFS 特定部分。该命令检查文件系统的一致性，修复发现的问题。它也重放日志，把提交的改动应用到文件系统元数据，如果由于日志重放而声明文件系统是干净的，就不会再采取进一步操作。如果文件系统不认为是干净的，这意味着由于某种原因没有完整和正确地重放日志，或者文件系统不能单靠重放日志来恢复到一致状态，那么，就对文件系统执行一遍完整检查。

当执行全部完整性检查时，检查/修复实用程序首要目的是要达到可靠的文件系统状态，以防止将来文件系统崩溃或故障，第二个目的就是面对崩溃时保存数据。这意味着为了达到文件系统的一致性，实用程序可能丢弃数据。具体而言，当实用程序在不做假设的情况下，无法获得所需信息以将结构上不一致的文件或目录恢复到一致状态时，就会废弃数据。当遇到不一致的文件或目录时，就废弃整个文件或目录，而不再试图保存任何部分。任何由删除受损目录所孤立起来的文件或子目录，都放在文件系统根下的 lost+found 目录中。

文件系统检查/修复实用程序重点考虑的因素之一是所需虚存数量。通常，这些实用程序所需的虚存数量由文件系统的大小决定，这是由于所需虚存主要用于跟踪文件系统中个别块的分配状态。随着文件系统增大，块的数量增多，用来跟踪这些块所需的虚存数量也随之增加。

JFS 检查/修复实用程序的设计区别在于其虚存需求由文件系统中文件和目录的数量（而不是由块的数量）所决定。对 JFS 检查/修复实用程序而言，每个文件或目录的虚存大约为每个文件或目录 32 字节，或者对于包含百万个文件和目录的文件系统而言，不论其文件系统大小，虚存需求都是大约 32 兆字节。如同所有其它的文件系统，JFS 实用程序需要跟踪块分配状态，但避免使用虚存方法，而是使用位于实际文件系统中的一小块保留工作区来实现。

2.5　Ext3

ext3 是一个非常可靠、健壮的高质量日志文件系统。虽然与 resierFS、XFS 和 JFS 相比，ext3 的可伸缩性具有局限性，但已经证明，在大多数服务器和工作站所执行的典型文件系统操作中，使用 ext3

不仅速度很快而且很容易调整。

Ext3 是一个非常全面的文件系统。ext3 很像 ext2，不会提供 ReiserFS 那样特别快的小文件性能，但也不会带来意外的性能或功能瓶颈。因为 ext3 基于 ext2 的代码，所以它的磁盘格式和 ext2 的相同。这意味着，一个干净卸装的 ext3 文件系统可以作为 ext2 文件系统毫无问题地重新挂装。

由于都使用相同的元数据，可以执行 ext2 到 ext3 文件系统的现场升级。通过升级一些关键系统实用程序、安装新的 2.4 内核，并在每个文件系统上输入单条 tune2fs 命令，就可以把现有的 ext2 服务器转换成日志记录 ext3 系统，甚至可以在 ext2 文件系统已安装的情况下进行这些操作。这种转换是安全的、可逆的与简单的，并且不会导致任何意外的性能急剧下降。与转换到 XFS、JFS 或 ReiserFS 不同，这种转换不需数据备份和从头创建文件系统。

除了与 ext2 兼容之外，ext3 还通过共享 ext2 的元数据格式继承了 ext2 的其它优点。譬如，ext3 用户可以使用一个稳固的 fsck 工具。相反，ReiserFS 的 fsck 还很幼稚，当脆弱的元数据真的出现时，对脆弱元数据的修复过程将是困难和危险的。

2.5.1　日志的记录方式

Ext3 处理日志记录的方式与 ReiserFS 和其它日志记录文件系统所用的方式迥异。

2.5.1.1　仅记录元数据日志

典型的日志记录文件系统（譬如 ReiserFS、XFS 和 JFS）对元数据有特别处理，但是对数据不够重视。使用 ReiserFS、XFS 和 JFS 时，文件系统驱动程序记录元数据，但不提供数据日志记录。使用仅元数据日志记录，文件系统元数据将会异常稳固，因而可能永远不需要执行彻底 fsck。然而，意外的重新引导和系统锁定可能会导致最近修改数据的明显毁坏。

举例来说，假设正在修改名为 /tmp/myfile.txt 的文件时，机器意外锁定，被迫需要重新引导。如果使用的是仅元数据日志记录文件系统，文件系统元数据将容易地修复，但是，存在一种明显的可能性：在将 /tmp/myfile.txt 文件装入到文本编辑器时，文件不仅仅丢失最近的更改，而且还包含许多乱码甚至可能是完全不可读的信息。

2.5.1.2　ext3 的日志记录方法

在 ext3 里，日志记录代码使用一个特殊的称为“日志记录块设备”层或 JBD 的 API。JBD 被设计成在任何块设备上实现日志的特殊目的。Ext3 通过“钩入（hooking_in）”JBDAPI 来实现其日志记录。例如，ext3 文件系统代码将正在执行的修改告知 JBD，并且还会在修改磁盘上包含的特定数据之前请求 JBD 的许可。通过执行这些操作，给予了 JBD 代表 ext3 文件系统驱动程序管理日志的适当机会。这是很好的安排，因为 JBD 是作为一个单独的、一般实体而开发的，将来它可以用于向其它文件系统添加日志记录能力。

关于 JBD 管理的 ext3 日志有一些巧妙的特性。其中之一是，ext3 的日志存储在一个索引节点中——基本上是个文件。能否看到这个位于 /.journal 的文件，取决于如何在文件系统上“启用 ext3”的。当然，通过将日志存储在索引节点中，ext3 可以向文件系统添加必要的日志，而不需要对 ext2 元数据进行不兼容扩展。这是 ext3 文件系统保持对 ext2 元数据，以及 ext2 文件系统驱动程序的向后兼容性的关键方式之一。

2.5.2　　JBD 的日志记录方法

有许多方法用于实现日志。例如，文件系统开发者可能会设计出一种日志，该日志存储在主机文件系统上需要修改的字节范围。这种方法的好处在于，日志能够以一种非常高效的方式存储许多对文件系统的微小修改，这是因为它只记录需要修改的个别字节，而不记录除此以外的任何信息。

JBD 使用另外一种（从某种意义来说是更好的）方法。JBD 存储完整的被修改的文件系统块本身，而不是记录必定会被更改的字节范围。ext3 文件系统驱动程序也使用这种方法，存储内存中被修改的块（大小为 1K、2K 或 4K）的完整副本，以跟踪暂挂的 IO 操作。开始，这看起来有点浪费。毕竟，包含已修改数据的完整块中还可能包含未修改的（已经在磁盘上）数据。

JBD 所使用的方法称为物理日志记录，这意味着 JBD 使用完整的物理块，作为实现日志的主要媒介。相反，只存储已修改的字节范围而非完整块的方法称为逻辑日志记录，这是 XFS 所使用的方法。因为 ext3 使用物理日志记录，所以 ext3 日志将具有比其它文件系统日志（例如，XFS 日志）更大的相对磁盘占用。但是，因为 ext3 在文件系统内部和日志中使用完整块，ext3 处理的复杂度比实现逻辑日志记录的要小。另外，完整块的使用允许 ext3 执行一些额外的优化，譬如，将多个暂挂的 IO 操作“压扁”到同一内存数据结构的单个块中。接下来，这种优化允许 ext3 将这多个更改在一次写操作中写到磁盘上，而不需要多次写操作。此外，因为文字块数据存储在内存中，这些内存数据在写到磁盘之前，不必或只需作很少更改，大大减少了 CPU 开销。

2.5.3　　数据保护

ext3 有两种确保数据和元数据完整性的方法。

最初，ext3 被设计用来执行完整数据和元数据日志记录。在这种方式下（称之为“data=journal”方式），JBD 将所有对数据和元数据的更改都记录到文件系统中。因为数据和元数据都被记录，JBD 可以使用日志将元数据和数据恢复到一致状态。完整数据日志记录的缺点是它可能会比较慢，但可以通过设置相对较大日志来减少性能损失。

最近，ext3 添加了一种新的日志记录方式，该方式提供完整日志记录的好处而不会带来严重的性能损失，它只对元数据进行日志记录。但是，ext3 文件系统驱动程序保持对与每个元数据更新对应的特殊数据块的跟踪，将它们分组到一个称为事务的实体中。当事务应用于适当的文件系统时，数据块首先被写到磁盘。一旦写入数据块，元数据将随后写入日志。通过使用这种技术（称为“data=ordered”），即使只有元数据更改被记录到日志中，ext3 也能提供数据和元数据的一致性。ext3 缺省使用这种方式。

2.5.4　　ext3 工具

由于 ext3 向下兼容 ext2，可以使用 e2fsprogs 中的工具，包括 mke2fs、e2fsck、resize2fs、tune2fs 等。在较新的 e2fsprogs 中，增加了 mke3fs、e3fsck 两个工具，创建 ext3 文件系统可以直接使用 mke3fs 或 mkfs -t ext3，也可以先把文件系统创建为 ext2 再升级为 ext3。

已有的 ext2 文件系统可以使用 tune2fs –j 命令升级为 ext3 系统。



若 ext2 文件系统已安装在系统中，则升级后该分区的根目录中会有 .journal 文件。



可以参阅《Red Flag Server 4.0 系统管理手册》3.3 节中关于 ext3 文件系统的介绍。

2.5.5　XFS、JFS、ReiserFS、ext3 比较

到目前为止，选择合适 Linux 文件系统并不是一件简单的事。这首先依赖于您的运行环境和任务特性。一般认为，在处理大文件时，XFS 是首选；如果要处理大量小文件，应该选择 Reiserfs。到目前为止 JFS 在性能方面表现一般，但在稳定性方面还是让人放心的。ext3 在兼容 ext2 方面无疑是最好的，而且也有不错的性能。

以下是 4 种文件系统的主要参数比较。

	EXT3	Reiser FS	XFS	JFS
文件系统大小	4 TB	16TB（4GB 块）	18,000PB	4PB（512B 块）
				32PB（4KB 块）
块大小	1KB—4KB	最大 64KB，目前固定为 4KB	512B—64KB	512，1024，2048，4096B
文件大小	2GB(kernel 2.2.x)	2 — 10PB （ 4GB 块）	9,0000PB	512TB（512B 块）
				4 PB（4KB 块）

第 3 章　配置软件 RAID

RAID（Redundant Array of Inexpensive Disks）称为独立磁盘冗余阵列。RAID 的基本想法是把多个便宜的小磁盘组合到一起，成为一个磁盘组，使性能达到或超过一个容量巨大、价格昂贵的磁盘。

软件 RAID 使您不必购买昂贵的硬件 RAID 控制器和附件就能极大地增强 Linux 磁盘的 IO 性能和可靠性。由于 Linux RAID 是用软件实现的，所以它灵活、速度快。使用软件 RAID，可以实现将几个物理磁盘合并成一个更大的虚拟设备，达到性能改进和数据冗余的目的。



《Red Flag Server 4.0 安装手册》中介绍了在安装过程中创建软件 RAID 设备的方法和步骤。

3.1　软件 RAID 级别

目前用于 Linux 2.4 内核的软件 RAID 支持以下级别：线性模式，RAID0，RAID1，RAID4 和 RAID5。

- **线性模式**

将两个或更多的磁盘组合到一个物理设备中，***磁盘不必具有相同的大小***。因为磁盘彼此之间是附加在一起的，所以写入 RAID 设备时将首先填满磁盘 0，然后是磁盘 1，以此类推。

该级别中没有冗余。如果一块磁盘出现故障，那么很可能会丢失所有数据。不过，因为文件系统只是丢失一个大的连续数据组块，所以可以非常幸运地恢复一些数据。

对于单独的读和写，读取和写入性能不会提高。但是如果几个用户同时使用这一设备，并且一个用户实际使用第一块磁盘，而另一个用户正访问刚好位于第二块磁盘上的文件。如果发生这种情况，那么将会带来性能的提高。

- **RAID 0**

也称为分带（stripe）模式。它与线性模式类似，只不过读取和写入是在设备上并行完成的。设备的大小应该大致相等。因为所有访问都是并行完成的，所以设备都是同等填充的。如果一个设备比其他设备大很多，那么在 RAID 设备中仍将使用额外的空间，但是在写入 RAID 设备的高端部分时，将只能访问那个更大的磁盘。这当然会降低性能。

与线性模式一样，RAID0 也没有冗余。与线性模式不同的是，如果驱动器出现故障，那么将无法恢复任何数据。如果从 RAID0 中取出一个驱动器，那么 RAID 设备将不仅丢失一个连续的数据块，而是整个设备上都将充满小的空洞。

因为读取和写入是在设备上并行完成的，读取和写入性能将会增加。这通常是运行 RAID0 的主要

原因。如果磁盘总线足够快时，可以非常接近 N*P MB／S。

- **RAID 1**

一个真正具有冗余的模式。RAID1 可以用于两个或多个磁盘，拥有 0 块或多块备用磁盘。这种模式在其他一些磁盘上保留一块磁盘信息的准确镜像。当然，这些磁盘的大小必须相等。如果一块磁盘大于其他磁盘，那么您的 IUlD 设备将具有最小磁盘的大小。

如果最多取出了 N－l 块磁盘（或出现故障），那么所有数据仍然保持不变。如果有备用的磁盘，而且系统（例如，SCSI 驱动程序或 IDE 芯片组等）在故障过程中没有被破坏，那么在检测到驱动器故障之后，会立即在一块备用磁盘上开始重建镜像。

RAID1 的写入性能比在一个单独的设备上稍差一些，这是因为写入数据的相同副本必须发送到阵列中的每一块磁盘上。读取性能通常更为糟糕，但在 2.4 内核中已经得到大大改进。

- **RAID 4**

这个 RAID 级别很少使用。它可以用在三块或更多的磁盘上。它在一个驱动器上保存奇偶校验信息，并以这种方式将数据写入 RAID0 中的其他磁盘，而不是完全镜像信息。因为一块磁盘是为奇偶校验信息保留的，所以阵列的大小是（N－l）*S，其中，S 是阵列中最小驱动器的大小。就像在 RAID-1 中那样，磁盘的大小应该相等，否则就必须接受上面的公式。（N－l）*S 中的 S 是阵列中最小驱动器的大小。

如果一个驱动器出现故障，那么可以使用奇偶校验信息来重建所有数据。如果两个驱动器出现故障，那么所有数据都将丢失。

不经常使用这个级别的原因是奇偶校验信息存储在一个驱动器上。每次写入其他磁盘时，都必须更新这些信息。因此，如果奇偶校验磁盘并不比其他磁盘快很多，那么它就将成为一个瓶颈。

- **RAID 5**

在希望结合大量物理磁盘并且仍然保留一些冗余时，RAID5 可能是最有用的 RAID 模式。RAID5 可以用在三块或更多的磁盘上，并使用 0 块或更多的备用磁盘。就像 RAID4 一样，得到的 RAID5 设备的大小是（N－1）*S。

RAID5 与 RAID4 之间最大的区别就是奇偶校验信息均匀分布在各个驱动器上，这样就避免了 RAID 4 中出现的瓶颈问题。如果其中一块磁盘出现故障，那么由于有奇偶校验信息，所以所有数据仍然可以保持不变。如果可以使用备用磁盘，那么在设备故障之后，将立即开始重建数据。如果两块磁盘同时出现故障，那么所有数据都会丢失。RAID5 可以经受一块磁盘故障，但不能经受两块或多块磁盘故障。

读取和写入性能通常会提高，但很难预测其提高程度。

- **何为备用磁盘？**

备用磁盘是在一块活动磁盘出现故障之前不参与 RAID 的磁盘。在检测出设备故障时，该设备将被标记为“错误”，并立即开始在第一块备用磁盘上重建它。

因此，备用磁盘向 RAID5 添加了极好的额外安全措施，而这本来是很难实现的（在物理上）。因为通过备用磁盘实现了所有冗余，所以这允许系统在某个设备出现故障的情况下运行一段时间。

3.2 RAID 设置

3.2.1 配置文件示例

- **线性模式**

系统中有两个或更多分区，这些分区的大小不一定相等（当然也可以相等），需要将它们彼此连接。

设置 /etc/raidtab 文件。假设需要以线性模式添加两块磁盘，配置文件的示例如下：

```
raiddev /dev/md0
        raid-level              linear
        nr-raid-disks           2
        chunk-size              32
        persistent-superblock   1
        device                  /dev/sda2
        raid-disk               0
        device                  /dev/sdb3
        raid-disk               1
```

这里不支持备用磁盘。如果一块磁盘出现故障，那么阵列也会出现故障。这里没有添加备用磁盘的信息。

运行以下命令创建阵列：

mkraid /dev/md0

这将初始化阵列，写入超级块并运行这个阵列。可以检查一下 proc/mdstat 文件，确认 RAID 阵列正在运行。

现在，可以在设备 /dev/md0 上创建一个文件系统了，并像对待其他任何设备一样装载它了。

- **RAID 0**

系统中有两个或更多设备，它们的大小大致相等，可以通过并行访问它们来结合它们的存储容量并提高其性能。

设置 /etc/raidtab 文件，说明想要进行的 RAID0 配置。下面是配置文件的示例：

```
raiddev /dev/md0
        raid-level              0
        nr-raid-disks           2
        persistent-superblock   1
        chunk-size              4
        device                  /dev/sda2
        raid-disk               0
        device                  /dev/sdb3
        raid-disk               1
```

与线性模式类似，这里也不支持备用磁盘。RAID0 没有冗余，因此当一块磁盘出现故障时，阵列也会随之出现故障。

同样，运行以下命令初始化阵列：

mkraid /dev/md0

这将初始化超级块并运行 md0 设备。查看一下 /proc/mdstat，可以看到设备正在运行。

现在，就可以格式化、装载和使用 /dev/md0 设备了。

- **RAID 1**

系统中有两个大小大致相同的设备，要使这两个设备彼此镜像，或者想要在多个磁盘中设置一个备用磁盘，如果一个活动设备出现故障，那么它就会自动成为镜像的一部分。

设置 /etc/raidtab 文件，其示例如下：

```
raiddev /dev/md0
        raid-level              1
        nr-raid-disks           2
        nr-spare-disks          0
        chunk-size              4
```

```
        persist-superblock    1
        device            /dev/sda2
        raid-disk         0
        device            /dev/sdb3
        raid-disk         1
```

如果有备用磁盘，则可以将它们添加到设备说明的最后：

```
    device /dev/sdc4
    spare-disk 0
```

一定要记住设置相应地 nr-spare-disks 项。

现在已经准备好初始化 RAID 了。

1、 必须重新创建镜像，如两个设备的内容必须同步（现在并不重要，因为还没有格式化设备）。
2、 使用以下命令：

    mkraid /dev/md0　　初始化镜像

3、 检查 /proc/mdstat 文件。它应该显示已经运行了 /dev/md0 设备，正在重建镜像，ETA 的重建已经完成。
4、 重建是使用空闲的 IO 带宽完成的。因此，虽然磁盘 LED 应该会不断闪烁，但是系统的响应性仍然不错。
5、 因为重建过程是透明的，所以实际上即使目前正在重建镜像，也可以使用这个设备。

- **RAID 5**

系统中有三个或更多大小大致相同的设备，想要将它们结合为一个较大的设备，但为了数据的安全性，仍然需要保留一定程度的冗余。最终您将有许多用作备用磁盘的设备，在其他设备出现故障之前，这些设备不会参与到阵列中。

如果使用了 N 个设备，其中最小设备的大小是 S，那么整个阵列的大小是（N－1）*S。丢失的空间用于奇偶校验（冗余）信息。因此，如果任何一块磁盘出现故障，那么所有数据都将保持不变。但如果两块磁盘出现故障，那么所有数据都将丢失。

按照以下的方法配置 /etc/raidtab 文件：

```
    raiddev /dev/md0
        raid-level        5
        nr-raid-disks     5
        nr-spare-disks    0
```

```
persistent-superblock    1
chunk-size               32
device                   /dev/sda2
raid-disk                0
device                   /dev/sdb3
raid-disk                1
device                   /dev/sdc4
raid-disk                2
device                   /dev/sdd5
raiddisk                 3
device                   /dev/sda6
raiddisk                 4
```

如果有任何备用磁盘，那么将以类似的方式插入这些磁盘，按照下面的 spare-disk 说明进行：

```
device /dev/sdb1
spare-disk 0
```

运行以下命令：

mkraid /dev/md0

如果顺利，那么磁盘将开始快速转动，因为它们将开始重建阵列。

如果成功创建了设备，那么重建过程就已经开始了。在重建完成之前，阵列是不一致的。但阵列是完全可以正常工作的（当然除了处理设备故障之外），您可以格式化它，即使在重建时也可以使用它。

3.2.2　启动和停止 RAID 设备

使用 raidstart 命令启动 RAID 设备，具体如下：

raidstart /dev/md0

在运行 RAID 设备时，可以使用如下命令停止它：

raidstop /dev/md0

3.2.3　几个有用的概念

- **持久的超级块**

最早时，raidtools 先要读取 /etc/raidtab 文件，然后初始化阵列。但是，这要求装载 /etc/raidtab 所在的文件系统。如果想要在 RAID 上启动，那么是很不适宜的。

此外，在 RAID 设备上装载文件系统时，原来的方法会使问题复杂化。它们不能像往常那样放入 /etc/fstab 文件，而是必须从初始化脚本装载。

持久的超级块解决了这些问题。如果在 /etc/raidtab 文件中使用 persist-superblock 选项，初始化阵列时，一个特殊的超级块被写入参与阵列的所有磁盘的开始位置。这允许内核从所涉及的磁盘上直接读取 RAID 设备的配置，而不是从某些配置文件中读取。

但是，仍然应该维护一个一致的 /etc/raidtab 文件，因为可能会在以后重建阵列时需要这个文件。

- **数据块大小（chunk-size）**

需要选择某个合适的块大小，它是指可以写入设备的最小基本数据量。块大小为 4KB 时，写入 16KB 记录将导致第一个和第三个 4KB 的块写入第一块磁盘，第二个和第四个块写入第二块磁盘（使用两块磁盘的 RAID0）。因此对于大型写入来说，可以看到，使用较大的块大小可以减少开销，而主要包含小文件的阵列使用较小的块大小会更有利。

必须为所有 RAID 级别（包括线性模式）指定块大小。但是，块大小对于线性模式来说没有任何区别。

要想得到最佳的性能，应该尝试这个值以及 RAID 阵列上的文件系统的块大小。



/etc/raidtab 中 chunk-size 的参数以 KB 为单位指定组块的大小。因此 4 表示 4KB。

3.3　RAID 不能做什么

RAID 的容错功能设计用于避免由偶发的驱动器故障所产生的负面影响。这种设计非常好。但是，对于各种各样的可靠性问题，RAID 并非总是理想的解决方案。在生产环境中，在实现具有容错功能的 RAID（1、4、5）之前，准确了解 RAID 能做什么及不能做什么至关重要。当处于依赖 RAID 的境况中时，我们不希望对它的作用抱有错误的认识。我们首先要澄清对 RAID 1、4 和 5 的一些常见错误认识。

许多人认为，如果将所有重要数据保存在 RAID 1、4 或 5 卷上，就没有必要再对这些数据执行定期的备份。这是完全错误的，理由如下：RAID 1/4/5 有助于避免由偶然的驱动器故障引起的意外停机；但是，它并不能防止意外或恶意的数据损坏。如果读者在 RAID 卷上以 root 身份键入 “cd /; rm -rf *”，那么顷刻之间将丢失大量重要的数据，对于这种情况，就算拥有一个包含 10 个驱动器的 RAID 5 配置也无济于事。同样，如果服务器失窃，或者建筑物失火，那么 RAID 也帮不上忙。毫无疑问，如果没有实施备份策略，就不会拥有历史数据的档案文件。假如某位同事删除了一批重要文件，也无法将它们恢

复。仅此一点就应该足以让您相信，在大多数情况下，即使是在考虑采用 RAID 1、4 和 5 的同时，也应该规划并实施一种备份策略。

在由劣质硬件组成的系统上实施软件 RAID 是另一种错误认识。如果正在装配一台要承担重要任务的服务器，那么在预算许可的范围之内购买质量最好的硬件是合理的。如果系统不稳定或者散热不良，那么将陷入一种 RAID 无能为力的困境。与此类似，如果停电，RAID 显然也不能提供更长的正常运行时间。如果服务器计划担负任何比较重要的任务，请确保已为它配备了不间断电源（UPS）。

文件系统存在于软件 RAID 卷之上，使用软件 RAID 并不能避开文件系统问题，例如，如果恰好在使用一种非日志文件系统或者定期整理碎片的文件系统，则可能存在耗时且易出问题的文件系统检查。因此，软件 RAID 不会提高 ext2 文件系统的可靠性；这就是为什么在 Linux 阵营中仍然强调保留 ReiserFS、JFS 和 XFS 的原因。软件 RAID 和可靠的日志文件系统是一种理想的组合。

第 4 章　LVM 使用手册

4.1　简介

4.1.1　什么是 LVM?

LVM 是 Logical Volume Manager（逻辑卷管理）的简写，它由 Heinz Mauelshagen 在 Linux 2.4 内核上实现。

与传统的磁盘与分区相比，LVM 为计算机提供了更高层次的磁盘存储。它使系统管理员可以更方便的为应用与用户分配存储空间。在 LVM 管理下的存储卷可以按需要随时改变大小与移除（可能需对文件系统工具进行升级）。LVM 也允许按用户组对存储卷进行管理，允许管理员用更直观的名称（如“sales”、“development”）代替物理磁盘名（如“sda”、“sdb”）来标识存储卷。

4.1.2　为什么使用 LVM?

LVM 通常用于装备有大量磁盘的系统，但它同样适于仅有一、两块硬盘的小系统。

4.1.2.1　小系统使用 LVM 的益处

传统的文件系统是基于分区的，一个文件系统对应一个分区。这种方式比较直观，但不易改变：

1、不同的分区相对独立，无相互联系，各分区空间很容易利用不平衡，空间不能充分利用。

2、当一个文件系统或分区已满时，无法对其扩充，只能重新分区、建立文件系统；或把分区中的数据移到另一个更大的分区中；或采用符号连接的方式使用其它分区的空间，非常麻烦。

3、如果要把硬盘上的多个分区合并在一起使用，只能采用再分区的方式，这个过程需要数据的备份与恢复。

当采用 LVM 时，情况有所不同：

1、硬盘的多个分区由 LVM 统一为卷组管理，可以方便的加入或移走分区以扩大或减小卷组的可用容量，硬盘空间被充分利用。

2、文件系统建立在逻辑卷上，而逻辑卷可在卷组容量范围内根据需要改变大小。

3、文件系统建立在 LVM 上，可以跨分区，方便使用。

4.1.2.2　大系统使用 LVM 的益处

在使用很多硬盘的大系统中，使用 LVM 主要是方便管理、增加了系统的扩展性。

在一个有很多不同容量硬盘的大型系统中，对不同用户的空间分配是一个技巧性的工作，要在用户需求与实际可用空间中寻求平衡。

用户或用户组的空间建立在 LVM 上，可以随时按要求增大，或根据使用情况对各逻辑卷进行调整。当系统空间不足而加入新的硬盘时，不必把用户的数据从原硬盘迁移到新硬盘，而只须把新的分区加入卷组并扩充逻辑卷即可。同样，使用 LVM 可以在不停止服务的情况下，把用户数据从旧硬盘转移到新硬盘空间中去。

4.2　LVM 的构成

LVM 的结构简图如下：

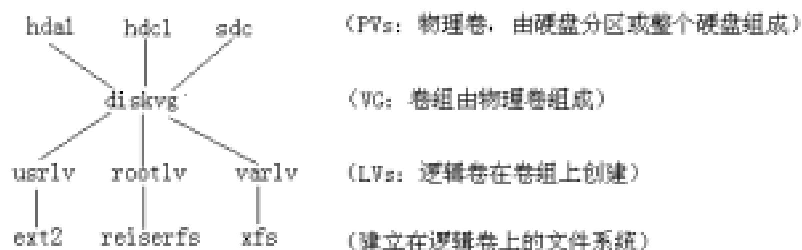


图4-1　　LVM 结构简图

- 卷组（VG）：volume group

  LVM 中最高抽象层，由一个或多个物理卷所组成的存储器池。

- 物理卷（PV）：physical volume

  典型的物理卷是硬盘分区，也可以是整个硬盘或已创建的软件 RAID 设备。

- 逻辑卷（LV）：logical volume

  相当于非 LVM 系统中的分区，它在卷组上建立，是一个标准的块设备，可以在其上建立文件系统。

- 物理块（PE）：physical extent

  物理卷以大小相等的“**块**”为单位存储，块的大小与卷组中逻辑卷块的大小相同。

- 逻辑块（LE）：logical extent

  逻辑卷以“**块**”为单位存储，在同一卷组中的所有逻辑卷的块大小是相同的。

4.2.1　总述

例子：有一个卷组 VG1，它的物理块大小为 4MB。在这个卷组中有 2 个硬盘分区：/dev/hda1 与 /dev/hdb1，它们分别成为物理卷 PV1 与 PV2。物理卷将按 4MB 为单位分块，如 PV1 与 PV2 分别可

分为 99 与 248 块。在 VG1 上建立逻辑卷，它的大小可在 1 至 347（99+248）块之间。当建立逻辑卷时，会建立逻辑块与物理块的一一映射关系。

4.2.2　映射模式（linear/striped）

在建立逻辑卷时，可以选择逻辑块与物理块映射的策略：

- **线性映射**

把一定范围的物理块按顺序分配给逻辑卷，如 LV 的 LE1–99 映射到 PV1，LE100–347 映射到 PV2。

- **交错模式**

将把逻辑块交错映射到不同的物理卷中，如 LV 的 LE1 映射为 PV1 的 PE1，LE2 映射为 PV2 的 PE1，LE3 映射为 PV1 的 PE2……。这种方式可以提高逻辑卷的性能，但是采用这种方式建立的逻辑卷将不能在它们所在的物理卷中扩展。

4.2.3　Snapshots（快照）

LVM 提供了一个非常好的特性：snapshots。它允许管理员建立一个块设备：该设备是一逻辑卷在某一时刻冻结的精确拷贝。这个特性通常用于批处理过程（如备份）需要处理逻辑卷，但又不能停止系统。当操作完成时，snapshot 设备可以被移除。这个特性要求在建立 snapshot 设备时逻辑卷处于相容状态。

4.3　LVM 的一般操作

4.3.1　建立 PV

为把一个磁盘或分区作为 PV，首先应使用 pvcreate 对其初始化，如对 IDE 硬盘 /dev/hdb。

- 使用整个磁盘

　　# **pvcreate /dev/hdb**

这将在磁盘上建立 VG 的描述符。

- 使用磁盘分区，如 /dev/hdb1。

使用 fdisk 的 t 命令把 /dev/hda1 的分区类型设为 0x8e，然后运行：

　　# **pvcreate /dev/hdb1**

这将在分区 /dev/hda1 上建立 VG 的描述符。

PV 初始化命令 pvcreate 的一般用法为：

pvcreate PV1 [PV2 ...]

它的参数可以是整个磁盘、分区，也可以是一个 loop 设备。

4.3.2 建立 VG

在使用 pvcreate 建立了 PV 后，可以用 vgcreate 建立卷组，如 PV1、PV2 分别是 /dev/hda1 与 /dev/hdb1，使用如下命令将建立一个名为 testvg 的卷组，它由两个 PV：/dev/hda1 与 /dev/hdb1 组成。

vgcreate testvg /dev/hda1 /dev/hdb1

vgcreate 的一般用法为：

vgcreate [options] VG_name PV1 [PV2 ...]

其中的可选项包括设置 VG 最大支持的 LV 数、PE 大小（缺省为 4MB）等。

注意：当使用 devfs 系统时，应使用设备的全名而不能是 Symbol Link，如对于上例，应为：

**vgcreate testvg /dev/ide/host0/bus0/target0/lun0/part1 **

/dev/ide/host0/bus0/target1/lun0/part1

4.3.3 激活 VG

在被激活之前，VG 与 LV 是无法访问的，这时可以使用以下命令激活所要使用的卷组。

vgchange -a y testvg

当不再使用 VG 时，可用如下命令使之不再可用。

vgchange -a n testvg

vgchange 可用来设置 VG 的一些参数，如是否可用（-a [y|n]选项）、支持最大逻辑卷数等。

4.3.4 移除 VG

在移除一卷组前应确认卷组中不再有逻辑卷，首先休眠卷组：

vgchange -a n testvg

然后可用 vgremove 移除该卷组：

```
# vgremove testvg
```

4.3.5　为 VG 增加新 PV

当卷组空间不足时，可以加入新的物理卷来扩大容量，这时可用命令 vgextend，如：

```
# vgextend testvg /dev/hdc1
```

其中 /dev/hdc1 是新的 PV，当然在这之前，它应使用 pvcreate 初始化。

4.3.6　从 VG 移除 PV

在移除 PV 之前，应确认该 PV 没用被 LV 使用，这可用命令 pvdisplay 查看，如：

```
# pvdisplay /dev/hda1
--- Physical volume ---
PV Name               /dev/hda1
VG Name               testvg
PV Size               1.95 GB / NOT usable 4 MB [LVM: 122 KB]
PV#                   1
PV Status             available
Allocatable           yes (but full)
Cur LV                1
PE Size (KByte)       4096
Total PE              499
Free PE               0
Allocated PE          499
PV UUID               Sd44tK-9IRw-SrMC-MOkn-76iP-iftz-OVSen7
```

如这个 PV 仍在被使用，则应把数据传移到其它 PV 上。在确认它未被使用后，可用命令 vgreduce 把它从 VG 中删除，如：

```
# vgreduce testvg /dev/hda1
```

4.3.7　创建 LV

在创建逻辑卷前，应决定 LV 使用哪些 PV，这可用命令 vgdisplay 与 pvdisplay 查看当前卷组与 PV 的使用情况。在已有的卷组上创建逻辑卷使用命令 lvcreate，如：

lvcreate -L1500 -ntestlv testvg

将在卷组 testvg 上建立一个 1500MB 的线性 LV，其命名为 testlv，对应的块设备为 /dev/testvg/testlv。

lvcreate -i2 -I4 -l100 -nanothertestlv testvg

将在卷组 testvg 上建立名为 anothertestlv 的 LV，其大小为 100LE，采用交错方式存放，交错值为 2，块大小为 4KB。

如果需要 LV 使用整个 VG，可首先用 vgdisplay 查找 Total PE 值，然后在运行 lvcreate 时指定，如：

vgdisplay testvg | grep “Total PE”

Total PE　　10230

lvcreate -l 10230 testvg -n mylv

将使用卷组 testvg 的全部空间创建逻辑卷 mylv。

在创建逻辑卷后，就可在其上创建文件系统并使用它。

命令 lvcreate 的常用方法：

lvcreate [options] -n 逻辑卷名 卷组名 [PV1 ...]

其中的常用可选项有：

-i Stripes：	采用交错（striped）方式创建 LV，其中 Stripes 指卷组中 PV 的数量。
-I Stripe_size：	采用交错方式时采用的块大小（单位为 KB），Stripe_size 必须为 2 的指数：2^N，N=2,3...9。
-l LEs：	指定 LV 的逻辑块数。
-L size：	指定 LV 的大小，其后可以用 K、M、G 表示 KB、MB、GB。
-s：	创建一已存在 LV 的 snapshot 卷。
-n name：	为 LV 指定名称。

4.3.8　删除 LV

为删除一个逻辑卷，必须首先从系统卸载其上的文件系统，然后可用 lvremove 删除，如：

umount /dev/testvg/testlv

lvremove /dev/testvg/testlv

lvremove -- do you really want to remove "/dev/testvg/testlv"? [y/n]: y

lvremove -- doing automatic backup of volume group "testvg"

lvremove -- logical volume "/dev/testvg/testlv" successfully removed

4.3.9　扩展 LV

为逻辑卷增加容量可用使用 lvextend，即可以指定要增加的尺寸也可以指定扩容后的尺寸，如：

lvextend -L12G /dev/testvg/testlv

lvextend -- extending logical volume "/dev/testvg/testlv" to 12 GB

lvextend -- doing automatic backup of volume group "testvg"

lvextend -- logical volume "/dev/testvg/testlv" successfully extended

将扩大逻辑卷 testlv 的容量为 12GB。

lvextend -L+1G /dev/testvg/testlv

lvextend -- extending logical volume "/dev/testvg/testlv" to 13 GB

lvextend -- doing automatic backup of volume group "testvg"

lvextend -- logical volume "/dev/testvg/testlv" successfully extended

将为 LV testlv 再增大容量 1GB 至 13GB。

为 LV 扩容的一个前提是：LV 所在的 VG 有足够的空闲存储空间可用。

在为 LV 扩容之后，应同时为 LV 之上的文件系统扩容，使二者相匹配。对不同的文件系统有相对应的扩容方法。

4.3.9.1　ext2/ext3

除非内核已有 ext2online 补丁，否则在改变 ext2/ext3 文件系统的大小时应卸载它：

umount /dev/testvg/testlv

resize2fs /dev/testvg/testlv

mount /dev/testvg/testlv /home

这里假设 testlv 安装点为 /home。在 es2fsprogs-1.19 或以上版本中包含 resize2fs 命令。

在 LVM 发行包中有一个称为 e2fsadm 的工具，它同时包含了 lvextend 与 resize2fs 的功能，如：

```
# e2fsadm -L+1G /dev/testvg/testlv
```

等价于下面两条命令：

```
# lvextend -L+1G /dev/testvg/testlv
# resize2fs /dev/testvg/testlv
```

但用户仍需首先卸载文件系统。

4.3.9.2 reiserfs

与 ext2 不同，Reiserfs 不必卸载文件系统，如：

```
# resize_reiserfs -f /dev/testvg/testvl
```

4.3.9.3 xfs

SGI XFS 文件系统必须在安装的情况下才可改变大小，并且要使用安装点而不是块设备，如：

```
# xfs_growfs /home
```

4.3.10 缩小 LV

逻辑卷可扩展同样也可缩小，但应在缩小 LV 之前首先减小文件系统，否则将可能导致数据丢失。

4.3.10.1 ext2/ext3

可以使用 LVM 的工具 e2fsadm 操作，如：

```
# umount /home
# e2fsadm -L-1G /dev/testvg/testvl
# mount /home
```

如果采用 resize2fs，就必须知道缩少后卷的块数：

```
# umount /home
# resize2fs /dev/testvg/testvl 524288
# lvreduce -L-1G /dev/testvg/testvl
# mount /home
```

4.3.10.2　　reiserfs

在缩小 reiserfs 时，应首先卸载它，如：

```
# umount /home
# resize_reiserfs -s-1G /dev/testvg/testvl
# lvreduce -L-1G /dev/testvg/testvl
# mount -treiserfs /dev/testvg/testvl /home
```

4.3.10.3　　xfs

无法实现。

4.3.11　在 PV 间转移数据

若要把一个 PV 从 VG 中移除，应首先把其上所有活动 PE 中的数据转移到其它 PV 上，而新的 PV 必须是本 VG 的一部分，有足够的空间。如要把 PV1:/dev/hda1 上的数据移到 PV2:/dev/sda1 上可用命令：

```
# pvmove /dev/hdb1 /dev/sdg1
```

如果在该 PV 之上的 LV 采用交错方式存放，则这个转移过程不能被打断。

建议在转移数据之前备份 LV 中的数据。

4.3.12　系统启动和关闭

为使系统启动时可自动激活并使用 LVM，可将以下几行添加到启动 rc 脚本中：

```
/sbin/vgscan
/sbin/vgchange -a y
```

这些行将浏览所有可用的卷组并激活它们。要注意的是，它们应在安装卷组上的文件系统操作之前被执行，否则将无法正常安装文件系统。

在系统关机时，要关闭 LVM，可以将以下这行添加到关机 rc 脚本中，并确保它在卸装了所有文件系统后执行：

```
/sbin/vgchange -a n
```

4.4　磁盘分区问题

4.4.1　一个磁盘上的多个分区

LVM 允许 PV 建立在几乎所有块设备上，如整个硬盘、硬盘分区、Soft RAID：

pvcreate /dev/sda1

pvcreate /dev/sdf

pvcreate /dev/hda8

pvcreate /dev/hda6

pvcreate /dev/md1

所以在一块硬盘上可以有多个 PV 或分区，但一般建议一块硬盘上只有一个 PV：

- 便于管理，易于处理错误。
- 避免交错方式中性能下降。LVM 不能辨别两个 PV 是否在同一硬盘上，故当采用交错方式时，会导致性能更差。

但在某些情况下可采用：

- 把已存在的系统合并到 LVM 中。在一个只有少数硬盘的系统中，转换为 LVM 时需在各分区之间转移数据。
- 把一个大硬盘分给不同的 VG 使用。

当一个 VG 的有不同的 PV 在同一硬盘时，创建交错方式的 LV 时应注意使用哪一个 PV。

4.4.2　Sun disk labels

仅在 SUN 的 SPARC 系统中有此问题。

4.5　建立 LVM 用例

在本节中，将在 3 块 SCSI 硬盘：/dev/sda，/dev/sdb，/dev/sdc 上按步建立 LVM。

4.5.1　准备分区

首先要做的是初始化硬盘，建立 PV。***这将会删除硬盘上的原有数据***。在此，使用整个硬盘为 PV：

pvcreate /dev/sda

pvcreate /dev/sdb

pvcreate /dev/sdc

pvcreate 在每个硬盘的起始端建立卷组描述区（volume group descriptor area，VGDA）。

4.5.2　创建卷组

利用上面三个 PV 建立卷组：

vgcreate test_vg /dev/sda /dev/sdb /dev/sdc/

然后可用 vgdisplay 查看和验证卷组的信息：

```
# vgdisplay
--- Volume Group ---
VG Name             test_vg
VG Access           read/write
VG Status           available/resizable
VG #                1
MAX LV              256
Cur LV              0
Open LV             0
MAX LV Size         255.99 GB
Max PV              256
Cur PV              3
Act PV              3
VG Size             1.45 GB
PE Size             4 MB
Total PE            372
Alloc PE / Size     0 / 0
Free  PE / Size     372/ 1.45 GB
VG UUID             nP2PY5-5TOS-hLx0-FDu0-2a6N-f37x-0BME0Y
```

其中最重要的前三条要正确，且 VS size 是以上三个硬盘容量之和。

4.5.3　建立 LV

在确定卷组 test_vg 正确后，就可在其上创建 LV。LV 的大小可在 VG 大小范围内任意选择，如同在硬盘上分区。

4.5.3.1　　建立线性方式 LV

在 test_vg 上建立一个大小为 1GB 的线性方式 LV：

```
# lvcreate -L1G -ntest_lv test_vg
lvcreate -- doing automatic backup of "test_vg"
lvcreate -- logical volume "/dev/test_vg/test_lv" successfully created
```

4.5.3.2　　建立交错方式 LV

在 test_vg 上建立一个大小为 1GB 的交错方式 LV，交错参数为 4KB：

```
# lvcreate -i3 -I4 -L1G -ntest_lv test_vg
lvcreate -- rounding 1048576 KB to stripe boundary size 1056768 KB / 258 PE
lvcreate -- doing automatic backup of "test_vg"
lvcreate -- logical volume "/dev/test_vg/test_lv" successfully created
```



如果使用 -i2 参数，则 LV 将仅使用 test_vg 中的两块硬盘。

4.5.4　建立文件系统

在 LV test_lv 创建后，就可在其上建立文件系统。

如，ext2/ext3 系统：

```
# mke2fs /dev/test_vg/test_lv
```

如，reiserfs：

```
# mkreiserfs /dev/test_vg/test_lv
```

4.5.5　测试文件系统

安装 LV：

```
# mount /dev/test_vg/test_lv /mnt
# df
Filesystem             1k-blocks   Used     Available   Use%  Mounted on
/dev/hda1              1311552     628824   616104      51%   /
/dev/test_vg/test_lv   1040132     20       987276      0%    /mnt
```

则可以通过 /mnt 访问 LV。

4.6　使用 snapshot 做备份

例如我们要对卷组 “test_vg” 每晚进行数据库备份，就要采用 snapshot 类型的卷组。这种卷组是其它卷组的一个只读拷贝，它含有在创建 snapshot 卷组时原卷组的所有数据，这意味可以备份这个卷组而不用担心在备份过程中数据会改变，也不需要暂时关闭数据库卷以备份。

4.6.1　建立 snapshot 卷

一个 snapshot 卷可大可小，但必须有足够的空间存放所有在本 snapshot 卷生存期间改变的数据，一般最大要求是原卷组的 1.1 倍。如空间不够，snapshot 卷将不能使用。

```
# lvcreate -L592M -s -n dbbackup /dev/test_vg/databases
lvcreate -- WARNING: the snapshot must be disabled if it gets full
lvcreate -- INFO: using default snapshot chunk size of 64 KB for  "/dev/test_vg/dbbackup"
lvcreate -- doing automatic backup of "test_vg"
lvcreate -- logical volume "/dev/test_vg/dbbackup" successfully created
```

4.6.2　安装 snapshot 卷

现在可以安装该卷：

```
# mkdir /mnt/test_vg/dbbackup
# mount /dev/test_vg/dbbackup /mnt/test_vg/dbbackup
mount: block device /dev/test_vg/dbbackup is write-protected, mounting read-only
```

从上面可以看出，snapshot 卷是只读的。

当使用 XFS 文件系统时，mount 命令要使用 nouuid 与 norecovery 选项：

```
# mount /dev/test_vg/dbbackup /mnt/test_vg/dbbackup -o nouuid,norecovery,ro
```

4.6.3　备份数据

如采用 tar 向磁带备份：

```
# tar -cf /dev/rmt0 /mnt/test_vg/dbbackup
```

4.6.4　删除 snapshot 卷

在完成备份后，就可卸载并删除 snapshot 卷。

```
# umount /mnt/test_vg/dbbackup
# lvremove /dev/test_vg/dbbackup
lvremove -- do you really want to remove "/dev/test_vg/dbbackup"? [y/n]: y
lvremove -- doing automatic backup of volume group "test_vg"
lvremove -- logical volume "/dev/test_vg/dbbackup" successfully removed
```

4.7 更换卷组硬盘

由于某种原因，需要用新的硬盘替代卷组中的旧硬盘，如用一个 SCSI 硬盘替换 IDE 硬盘，其步骤如下。

4.7.1 准备和初始化新硬盘

首先用 pvcreate 命令初始化新的硬盘，如使用整个硬盘：

```
# pvcreate /dev/sdf
pvcreate -- physical volume "/dev/sdf" successfully created
```

4.7.2 加入卷组

把新硬盘加入卷组：

```
# vgextend test_vg /dev/sdf
vgextend -- INFO: maximum logical volume size is 255.99 Gigabyte
vgextend -- doing automatic backup of volume group "test_vg"
vgextend -- volume group "test_vg" successfully extended
```

4.7.3 数据搬家

在移除旧硬盘前，要把其上的数据转移到新硬盘上。在转移数据时，不要求卸载文件系统，但建议在数据转移前进行备份，以防转移过程中出现意外导致数据丢失。

pvmove 用来实现数据转移，根据数据量的多少，可能要使用大量的时间，并可能降低逻辑卷的性能，因此要在系统不太忙时执行操作。

```
# pvmove /dev/hdb /dev/sdf
pvmove -- moving physical extents in active volume group "test_vg"
```

pvmove -- WARNING: moving of active logical volumes may cause data loss!

pvmove -- do you want to continue? [y/n] y

pvmove -- 249 extents of physical volume "/dev/hdb" successfully moved

4.7.4　移除未用硬盘

当数据被转移到其它硬盘后，就可以从卷组中删除这块不再使用的硬盘了。

vgreduce dev /dev/hdb

vgreduce -- doing automatic backup of volume group "test_vg"

vgreduce -- volume group "test_vg" successfully reduced by physical volume:

vgreduce -- /dev/hdb

从此，卷组 test_vg 不再使用 IDE 硬盘 /dev/hdb，这块硬盘可以从机器中拆下或用作其它用途。

4.8　迁移卷组到其它系统

把一个卷组转移到其它系统是很容易的（如更换服务器），这要用命令 vgexport 与 vgimport。

4.8.1　卸载文件系统

为整体搬迁卷组，应首先把它从文件系统中卸载，如：

unmount /mnt/design/users

4.8.2　设置卷组为非活动状态

把卷组从内核中卸载，以避免任何对它可能的操作：

vgchange -an test_vg

vgchange -- volume group "test_vg" successfully deactivated

4.8.3　Export 卷组

这个操作不是必须的，但它可以防止系统对卷组的访问：

vgexport test_vg

vgexport -- volume group "test_vg" sucessfully exported

当机器关机后，构成卷组的硬盘就可被转移到新的服务器上。

4.8.4　Import 卷组

在新的服务器上，可用 pvscan 查看卷组情况，如在这台计算机上，新的硬盘设备为 /dev/sdb，使用 pvscan 可以：

```
# pvscan
pvscan -- reading all physical volumes (this may take a while...)
pvscan -- inactive PV "/dev/sdb1"    is in EXPORTED VG "test_vg" [996 MB / 996 MB free]
pvscan -- inactive PV "/dev/sdb2"    is in EXPORTED VG "test_vg" [996 MB / 244 MB free]
pvscan -- total: 2 [1.95 GB] / in use: 2 [1.95 GB] / in no VG: 0 [0]
```

现在可以 import 卷组 test_vg（同时也激活它）以安装其上的文件系统。

```
# vgimport test_vg /dev/sdb1 /dev/sdb2
vgimport -- doing automatic backup of volume group "test_vg"
vgimport -- volume group "test_vg" successfully imported and activated
```

4.8.5　安装文件系统

```
# mkdir -p /mnt/design/users
# mount /dev/test_vg/users /mnt/design/users
```

在完成以上操作后，原卷组在新的服务器上就可使用了。

4.9　分割卷组

这种情况是：需要在系统中加入新的卷组，但没有其它可用新硬盘，而已有的卷组中还有大量空间可用。如向系统加入一个“design”卷组。

4.9.1　检查可用空间

```
# pvscan
pvscan -- reading all physical volumes（this may take a while...）
pvscan -- ACTIVE     PV "/dev/sda"    of VG "dev"      [1.95 GB / 0 free]
pvscan -- ACTIVE     PV "/dev/sdb"    of VG "sales" [1.95 GB / 1.27 GB free]
pvscan -- ACTIVE     PV "/dev/sdc"    of VG "ops"      [1.95 GB / 564 MB free]
```

pvscan -- ACTIVE　PV "/dev/sdd"　of VG "dev"　[1.95 GB / 0 free]

pvscan -- ACTIVE　PV "/dev/sde"　of VG "ops"　[1.95 GB / 1.9 GB free]

pvscan -- ACTIVE　PV "/dev/sdf"　of VG "dev"　[1.95 GB / 1.33 GB free]

pvscan -- ACTIVE　PV "/dev/sdg1"　of VG "ops"　[996 MB / 432 MB free]

pvscan -- ACTIVE　PV "/dev/sdg2"　f VG "dev"　[996 MB / 632 MB free]

pvscan -- total: 8 [13.67 GB] / in use: 8 [13.67 GB] / in no VG: 0 [0]

我们决定把 /dev/sdg1 与 /dev/sdg2 分配组 design，但首先要把其上的物理块移到其它卷的空闲空间中（如把卷组 dev 移到 /dev/sdf，卷组 ops 移到 /dev/sde）。

4.9.2　从选定硬盘移出数据

由于硬盘上的逻辑卷仍在使用，所以首先要转移它们的数据。

把所有在使用的物理块从 /dev/sdg1 上转移到 /dev/sde，从 /dev/sdg2 转移到 /dev/sdf。

pvmove /dev/sdg1 /dev/sde

pvmove -- moving physical extents in active volume group "ops"

pvmove -- WARNING: moving of active logical volumes may cause data loss!

pvmove -- do you want to continue? [y/n] y

pvmove -- doing automatic backup of volume group "ops"

pvmove -- 141 extents of physical volume "/dev/sdg1" successfully moved

pvmove /dev/sdg2 /dev/sdf

pvmove -- moving physical extents in active volume group "dev"

pvmove -- WARNING: moving of active logical volumes may cause data loss!

pvmove -- do you want to continue? [y/n] y

pvmove -- doing automatic backup of volume group "dev"

pvmove -- 91 extents of physical volume "/dev/sdg2" successfully moved

4.9.3　创建新卷组

现在把 /dev/sdg2 从卷组 dev 中分割出来，并加入到新卷组 design 中。我们可用 vgreduce 与 vgcreate 完成工作，但 vgsplit 此时更方便：

vgsplit dev design /dev/sdg2

```
vgsplit -- doing automatic backup of volume group "dev"
vgsplit -- doing automatic backup of volume group "design"
vgsplit -- volume group "dev" successfully split into "dev" and "design"
```

4.9.4　移除剩余的卷

接下来的工作把 /dev/sdg1 从卷组 ops 中分出来，并加入卷组 design：

```
# vgreduce ops /dev/sdg1
vgreduce -- doing automatic backup of volume group "ops"
vgreduce -- volume group "ops" successfully reduced by physical volume:
vgreduce -- /dev/sdg1

# vgextend design /dev/sdg1
vgextend -- INFO: maximum logical volume size is 255.99 Gigabyte
vgextend -- doing automatic backup of volume group "design"
vgextend -- volume group "design" successfully extended
```

4.9.5　建立新逻辑卷及文件系统

在卷组 design 上建立逻辑卷，为今后的方便，现只使用一部分空间：

```
# lvcreate -L750M -n users design
lvcreate -- rounding up size to physical extent boundary "752 MB"
lvcreate -- doing automatic backup of "design"
lvcreate -- logical volume "/dev/design/users" successfully created

# mke2fs /dev/design/users
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
96384 inodes, 192512 blocks
```

```
9625 blocks (5.00<!-- ) reserved for the super user
First data block=0
6 block groups
32768 blocks per group, 32768 fragments per group
16064 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

# mkdir -p /mnt/design/users
# mount /dev/design/users /mnt/design/users/
```

现在就可使用卷组 design。为方便使用，可把下面一行加入文件 /etc/fstab 中。

```
/dev/design/user    /mnt/design/users    ext2    defaults    1    2
```

4.10 转变根文件系统为 LVM



强烈要求在进行下面的操作前对系统进行备份。另外，把 ／ 文件系统建立在 LVM 上会导致系统升级很复杂。

在下面的例子中，系统除了 /boot 外都安装在同一个分区中，文件系统的情况为：

```
/dev/hda1    /boot
/dev/hda2    swap
/dev/hda3    /
```

进行转换的一个必要条件是硬盘上还有足够的空间给分区 /dev/hda4 创立 LVM 并把 / 分区的内容都复制到 LVM 上，否则：

1、 / 分区还有至少一半空间空闲，可以缩减 / 分区，并把分出的空间划分到分区 /dev/hda4；

可以使用 parted 工具方便的操作分区（参见***《Red Flag Server 4.0 系统管理手册》***），它可以缩减带有文件系统的分区。同样也可用 resize2fs 与 fdisk 工具完成这个操作；

2、 如果硬盘上已经没有足够的空间，就必须使用第二块硬盘，如 /dev/hdb。

在完成以上准备及备份系统后，可继续以下步骤：

1、 确认使用的 Linux 内核支持 LVM，并且在编译时设置了 CONFIG BLK DEV RAM 与 CONFIG BLK DEV INITRD。

2、 设置 /dev/hda4 分区类型为 LVM（8e）

　# **fdisk /dev/hda**

　Command (m for help): t

　Partition number (1-4): 4

　Hex code (type L to list codes): 8e

　Changed system type of partition 4 to 8e (Unknown)

　Command (m for help): w

3、 设置 LVM

- 初始化 LVM（vgscan）

  # **vgscan**

- 转变分区为 PV：

  # **pvcreate /dev/hda4**

- 建立卷组：

  # **vgcreate vg /dev/hda4**

- 建立逻辑卷用以存放根系统：（这里假设空间为 250MB）

  # **lvcreate -L250M root vg**

4、 在逻辑卷上建立文件系统并把系统复制到其上：

　# **mke2fs /dev/vg/root**

　# **mount /dev/vg/root /mnt/**

　# **find / -xdev | cpio -pvmd /mnt**

5、 修改新系统的 fstab 文件 /mnt/etc/fstab，使 / 安装到 /dev/vg/root：

　/dev/hda3　/　ext2　defaults　1　1

改变为：

　/dev/vg/root　/ ext2　defaults　1　1

6、 创建 LVM 初始化 RAM 盘。

　# **lvmcreate_initrd**

此处要确认为 lvmcreate_init 给出正确的 initrd image 文件名，它应在 /boot/ 目录下。

7、 在 /etc/lilo.conf 中为 LVM 加入新入口项，其形式如下：

```
image = /boot/KERNEL_IMAGE_NAME
label    = lvm
root      = /dev/vg/root
initrd    = /boot/INITRD_IMAGE_NAME
ramdisk = 8192
```

此处 KERNEL IMAGE NAME 是支持 LVM 的内核，INITRD IMAGE NAME 指由 lvmcreate_initrd 建立的 initrd image。如果 LVM 的配置很多，可以把 ramdisk 设置的大一些：此处为 8192，缺省为 4096。在 lvmcrate_initrd 的输出中有如下一行：

lvmcreate_initrd -- making loopback file（6189 kB）

其中括号中的数值为实际所需大小。

8、 运行 LILO，设置 BOOT 扇区：

lilo

9、 重启计算机，在 LILO 提示符处输入 “lvm”，启动计算机。此时系统的根文件系统是新建立的逻辑卷。此后可在 LILO 配置文件 /etc/lilo.conf 中加入以下一行：

default=lvm

并运行 lilo 设置缺省启动项为 lvm。

如果系统未能正常启动，可能的原因是内核不支持 LVM、initrd image 不正确等等。

10、在正常启动后，就可把硬盘其它分区：/dev/hda3 加入 LVM。

- 首先设置分区类型为 8e（LVM）

  # **fdisk /dev/hda**

  Command (m for help): t

  Partition number (1-4): 3

  Hex code (type L to list codes): 8e

  Changed system type of partition 3 to 8e (Unknown)

  Command (m for help): w

- 把它初始化为 PV，并加入卷组中。

  # **pvcreate /dev/hda3**

  # **vgextend vg /dev/hda3**

4.11 共享 LVM 卷

在使用 fibre-channel 或 shared-SCSI 的环境中，多台计算机以物理方式直接访问一组硬盘，于是可以使用 LVM 把这些硬盘分为不同的逻辑卷。如果需要共享数据，则应使用 GFS。



LVM 不支持物理共享访问，这会导致数据的丢失。

4.12 参考文献

AJ Lewis，LVM HOWTO，http://tldp.org/HOWTO/LVM-HOWTO/ 。

第 5 章　使用 EVMS

EVMS（Enterprise Volume Management System）称为企业卷管理系统，它把各方面的卷管理技术，如磁盘分区、Linux 逻辑卷管理（LVM）、multi-disk（MD）管理、OS2 和 AIX 卷管理和文件系统操作统一在单个的包中。EVMS 同时为 Linux 下的所有存储技术提供了统一的、可扩展的、基于插件的 API，EVMS 架构采用的插件模式更使得不同的卷管理方法可以被容易地扩展和定制。

5.1　EVMS 入门

本节描述了 EVMS 中经常使用的术语，同时说明 EVMS 中的层定义以及目前可以使用的 EVMS 用户界面。

5.1.1　一般术语

下表列出了 EVMS 中的常用术语。

扇区	块设备寻址的最低级别，它与在其它管理系统中所看到的标准含义一致。
存储对象	EVMS 中的所有存储结构，它能形成块设备，是一系列规则的扇区。
逻辑磁盘	一系列表现为物理设备的规则的相邻扇区。
磁盘段	一系列存在于逻辑磁盘或其它磁盘段的自然相邻的规则扇区。一个段的普通类推应是一个传统的磁盘分区，如 DOS 或 OS/2。
存储区域	一系列逻辑上相邻的规则扇区（没必要自然连续）。基本映射可以成为逻辑磁盘、段或其它区域。Linux LVM 和 AIX LVM LVs，以及 MD 设备，表现为 EVMS 中的区域。
存储容器	存储对象的集合。存储容器提供从这个集合到容器输出的一系列新存储对象的重映射。一个存储容器的适当类推应是卷组，如： AIX LVM 和 Linux LVM 中的卷组。但是，EVMS 容器不限制任何重新映射模式，这与在 LVM 或 AIX 中的卷组的情况一样。重映射是完全随意的。
特征对象	通过对 EVMS 本地特征的使用，将一个或多个磁盘、段、区或者其它性能对象创建为一个逻辑上连续的地址空间。
EVMS 逻辑卷	一个可安装的存储对象。EVMS 卷包括基本对象的末尾处的元数据，并且至少有一个静态名称和静态从号。EVMS 中的任何对象都可以转换为一个 EVMS 卷。
兼容性逻辑卷	一个不包括 EVMS 本地元数据的、可安装的存储对象。许多 EVMS 的插件提供对其它卷管理模式兼容性的支持。由于特指为 “兼容性” 的卷不包括任何 EVMS 本地元数据，则它一定是向后兼容至特殊模式。所有磁盘、段或区可以成为一个兼容性卷。但是，性能对象不能成为兼容性卷，只能成为 EVMS 卷。

5.1.2 层定义

EVMS 定义了一个分层结构，此结构中的每层插件向下创建抽象层。EVMS 也允许多数插件在同一层中创建抽象对象。下表以从大到小的次序给出了这些层的定义。

- **设备管理器**

第一层是逻辑设备管理器。这些插件与硬件设备驱动程序相联，创建第一个 EVMS 对象。目前，所有本地设备（大多数 IDE 和 SCSI 磁盘）都由单个插件控制。今后 EVMS 版本可能由另外的设备管理器来做网络设备管理，如存储区域网络（SAN）中的磁盘。

- **段管理器**

第二层是段管理器。一般来说，这些插件操作磁盘驱动的段或分区。引擎组件可以取代分区程序，如 **fdisk** 和 **disk druid**，核心组件也可以取代核心磁盘分区代码。段管理器也可以成为 “堆栈”，意味着一个段管理器可以从另一个段管理器中获得输入。

目前，该层中有 3 个插件。通常使用最多的是 DOS 段管理器。该插件控制 DOS 分区模式，该模式是 Linux 通常使用的模式。该插件也处理一些在使用 OS/2 分区时出现的特殊情况。

GUID 分区表（GPT）段管理器控制 IA-64 机器中的新 GPT 分区模式。Inter 的 “可扩展固件接口规格” 要求固件能够发现分区并产生适应磁盘分区的逻辑设备。由于全球唯一标识符标记（GUID）的广泛使用，在此规范中，分区模式称为 GPT。GUID 是一个 128 位长的标识符，故也称其为通用唯一标识符（UUID）。如 Intel Wired For Management Baseline 规格中所描述的那样，GUID 是时间域与空间域的组合，它能产生在整个 UUID 空间中的唯一标识符。这些标识符被广泛用于标记整个磁盘和单独区域的 GPT 分区的磁盘。GPT 分区磁盘具有以下几个功能，如：

- 保留元数据的源文件和备份文件；
- 通过允许多个分区，替换 MSDOS 分区嵌套；
- 使用 64 位逻辑块进行寻址；
- 使用 GUID 描述符标出分区和磁盘。

第三个插件控制 S/390 分区（CDL/LDL/CMS），此插件仍在开发中。目前，它支持发现、I/O 路径和段的创建及删除。

其它段管理器插件可以添加，以支持其它分区模式的功能（例如：Macintosh、Sun、和 SGI）。

- **区域管理器**

第三层是区域管理器。该层将为插件提供空间，确保这些插件与 Linux 或其它操作系统中的现有卷管理模式互相兼容。区域管理器将模拟在磁盘或分区上提供逻辑抽象的系统。

如同段管理器一样，区域管理器也可以成为堆栈。因此，区域管理器中的输入对象可以成为磁盘、段或其它区域。

目前，在 EVMS 中有四个区域管理器插件。第一个是 LVM 插件，它提供与 Linux LVM 的兼容性并

允许创建卷组或卷容器以及逻辑卷或逻辑区域。

另外两个的插件是 AIX 和 OS/2 区域管理器。AIX LVM 在功能性上与 Linux LVM 十分相似，它使用卷组和逻辑卷。AIX 插件仍在开发中。目前，它提供最重要的核心功能性，但在用户空间的使用仍然被限制。OS/2 插件提供与在 OS/2 下创建的卷的兼容性，不同于 Linux 和 AIX LVMs，OS/2 LVM 基于磁盘分区的线性连接，也基于不良扇区的重定位。

第四个区域管理器插件是 RAID 的 multi-disk（MD）插件。该插件提供软件 RAID 中的线性 RAID、0、1、4 和 5 等级。堆栈区域管理器接受 RAID 和 LVM 的组合。例如，一个磁盘条带集（RAID 0）可以作为 PV 在 LVM 中使用，或使用 RAID 1 反映两个 LVM LV。

- **EVMS 特征**

下一层是 "**EVMS 特征**"。该层是实施新本地 EVMS 特征功能的层。EVMS 特征可以建立于系统中任何一个对象上，包括磁盘、段、区域、或其它特征对象。EVMS 特征共享一个共同的元数据类型，该元数据更有效地发现特征对象，更可靠地恢复受损特征对象。

目前，EVMS 中有三个特征。第一个特征是驱动器连接。该插件允许任何数量的对象线性的连接到单个对象中。

第二个特征是不良（存储）块重定位（BBR）。BBR 监控其 I/O 路径并探测写入故障（这可能由受损磁盘导致的）。发生这样的故障时，该请求中的数据保存在一个新的位置。BBR 保留重映射的轨迹，并且在此位置保留改变其路径到新位置的所有附加的 I/O。

第三个特征是快照。快照提供了一种机制，不必使卷处于脱机状态就可立即创建一个卷的 "冻结" 副本。这十分有助于完成运行系统中的备份。所有卷都有快照过程（EVMS 或兼容性），并可以利用其它所有可用的对象作为一个辅助存储器。创建快照之后， "原始" 卷的写入导致该位置的原始内容被复制到快照的储存对象中。所以，保存在快照卷中的 I/O 如同来自于快照创建时的原始卷中。

- **文件系统接口模块**

文件系统接口模块（FSIM）是仅存于用户空间引擎中的 EVMS 的一个层。在一些卷管理操作中，这些插件经常用来协调文件系统。例如，当扩大或缩小一个卷时，也必须将文件系统扩大或缩小到适当的大小。该例子的顺序也是很重要的；一个文件系统不能在卷扩大之前扩大，而且一个卷也不能在文件系统缩小之前缩小。FSIM 允许 EVMS 以确保这种协调作用和顺序。

FSIM 也提供从 EVMS 用户接口之一执行文件系统的能力。例如，一个用户可以与 FSIM 相配合，创建新的文件系统和检查现有文件系统。

5.1.3　EVMS 用户界面

EVMS 目前有三种用户界面可供选用：用 GTK+ 编写的图形化管理界面——evmsgui、基于 ncurses 的界面和命令行界面（CLI）。GUI 界面友好，适用于所有用户；如果没有 GTK 库或 X 窗口，可以使用 ncurses 界面，如果是高级用户，则可以使用命令行完成工作。



为了避免与 RAID 或 LVM 产生冲突，EVMS 不会在系统启动时自动启动，如果要使用 EVMS，需要使用如下命令：

```
# /etc/rc.d/init.d/evms start
```

5.2 使用 EVMS

5.2.1 设备文件及 EVMS 名称空间

为了使用户空间程序接入 EVMS 卷，在 /dev/evms 路径下为每个输出卷创建设备文件。因为每次对引擎提交更改时都创建设备文件，所以总可以在 /dev/evms 路径下看见卷配置的当前状况。

各种名称空间存在于 /dev/evms 路径下。每个插件定义并保留其自己的名称空间。大多数普通的名称将是 DOS 段。在非 EVMS 的 Linux 系统中，磁盘驱动和驱动分区在 /dev 下以文件格式出现，本地设备管理器和 DOS 段管理器将这些相同的命名保留在 EVMS 中，将相应的设备文件创建在 /dev/evms 下。常见命名举例如下：

/dev/hda → /dev/evms/hda

/dev/hda1→/dev/evms/hda1

/dev/sdb → /dev/evms/sdb

/dev/sdb5→/dev/evms/sdb5

EVMS 特征创建有永久命名的逻辑卷，这些卷名常在 /dev/evms 下创建适当的设备文件。例如，如果创建一个 EVMS 快照并将其命名为 “Thursday_Snapshot”，那么，它将以 /dev/evms/Thursday_Snapshot 的形式出现。

Linux LVM 插件是使用路径 /dev/evms/lvm 的特殊插件名称空间的一个例子。在该路径中，为当前每个卷组创建附加路径，在适当的组路径下创建代表 LVM 逻辑卷的设备文件。举例如下：

/dev/Group1/Volume1→/dev/evms/lvm/Group1/Volume1

其它插件也存在名称空间。目前，OS/2 LVM 在 /dev/evms/os2 下，AIX LVM 在 /dev/evms/aix 下。

5.2.1.1 devfs

0.2.2 版本中，EVMS 支持核心设备文件系统（devfs）。如果在核心中启动 devfs 并将其安装在文件系统树中，就可以看见立即引导的卷设备文件。因为 devfs 确保将从号 EVMS 动态的分配到兼容性卷，该卷不再与这些卷的设备文件的从号同步，所以 Devfs 将是运行 EVMS 的首选方式。

5.2.1.2 固定 EVMS 设备节点

在没有运行 devfs 的系统中，/dev/evms 路径下的设备节点可以不与由 EVMS 核心输出的卷相一致。如果没有使用 devfs，以下列表详细列出了可能出现的潜在问题：

- 输出的卷可能不存在节点；
- 节点可能有与其相关的错误从号；
- 存在不再由 EVMS 核心输出的卷节点。

使用用户界面可以减少以上列出的许多问题。如果使用用户界面并对其提交改变，在 /dev/evms 下 EVMS 引擎将更新设备节点，因此该节点接受由 EVMS 核心输出的卷。

无需用户界面的启动消耗，evms_devnnode_fixup 程序通过更新设备节点提供了在 /dev/evms 路径下固定设备节点的方法。

通过指定 –d 选项，evms_devnode_fixup 可以像 daemon 一样运行。在 daemon 模式下，evms_devnode_fixup 首先在 /dev/evms 路径下固定设备节点。然后循环，等待 EVMS 运行时间的卷更改通知。在每个通知中，evms_devnode_fixup 在 /dev/evms 路径下固定设备节点。

5.2.2　使用有虚拟磁盘映像的 EVMS

如果要把 EVMS 核心插件构建为模块并在 EVMS 卷中安装根文件系统，请添加支持虚拟磁盘映像的 EVMS。如果没有虚拟磁盘映像，创建虚拟磁盘映像的一般指令是在文档文件 /initrd.txt 中的核心源码树下，一些分类包括了对创建虚拟磁盘映像的应用程序（视系统具体分区情况而定）。如果不想使用虚拟磁盘映像，应该把所有 EVMS 支持直接编入核心。

如果要修改系统的虚拟磁盘映像，请使用 gunzip 将其解压缩并通过一个环路（loopback）设备来安装。通过修改虚拟磁盘映像执行以下步骤。以下命令是基于在 /mnt/loop 下安装虚拟磁盘映像的设想。

当虚拟磁盘映像已经安装，请将 EVMS 核心模块复制其中。

mkdir -p /mnt/loop/lib/modules/x.y.z/kernel/drivers/evms

**cp /lib/modules/x.y.z/kernel/drivers/evms/*.o **

/mnt/loop/lib/modules/x.y.z/kernel/drivers/evms



在以前的代码中，x.y.z 是核心版本。

将 evms_rediscover 应用程序复制到虚拟磁盘映像中。如果没有使用 devfs，需要同时将 evms_devnode_fixup 应用程序进行复制。

cp /usr/local/sbin/evms_rediscover /mnt/loop/bin

cp /usr/local/sbin/evms_devnode_fixup /mnt/loop/bin



对 evms_rediscover 和 evms_devnode_fixup 进行静态编译，而且并不需要其它任何添加在虚拟磁盘映像中的动态库。

在虚拟磁盘映像中编辑 linuxrc 脚本，那么它会加载所有的 EVMS 核心模块。

命令一个核心重新查找，然后进行 evms_devnode_fixup（如果没有运行 devfs）。

将下列代码行添加到 linuxrc 脚本中。这些代码行应在安装根文件系统之前添加到 linuxrc 脚本中，那么根卷将被发现并将其利用。

```
insmod evms
insmod evms_passthru
insmod ldev_mgr
insmod dos_part
```

对在以上步骤中复制的所有 EVMS 模块添加 insmod 命令。

evms_rediscover

evms_devnode_fixup　　　　　　　　# 如果没有运行 devfs。

如果已经将插件构建为模块，且不打算安装使用 EVMS 的根文件系统，仅在 init scripts 添加入口来加载所有必需的 EVMS 模块。一旦已经编辑了脚本，则运行 evms_rediscover 应用程序。应在 /etc/fstab 文件运行之前尽早在 boot scripts 中运行重新查询。

5.3　EVMS GUI

EVMS GUI 提供了一个灵活且易于使用的图形化配置工具来管理卷和存储对象。EVMS GUI 扩大了用于前端接口的 Engine API 的用途。因为 Engine API 提供了抽象概念，所以 EVMS GUI 能够适应新的插件和特征而不需要改变代码。

5.3.1　使用 GUI 完成任务

需要进入 KDE 桌面环境，以 root 权限启动和运行 EVMS GUI 配置工具。可以采用以下方法启动：

1、　在系统主菜单中选择 “**系统→控制面板**”，打开控制面板，在 “**系统配置**” 标签页中，双击 “**EVMS 管理工具**”；

2、　在系统主菜单中选择 “**管理工具→EVMS 管理工具**”；

3、　在运行命令窗口或 shell 提示符下直接键入 evmsgui 。

下图 5-1 所示为 evmsgui 的配置主界面。

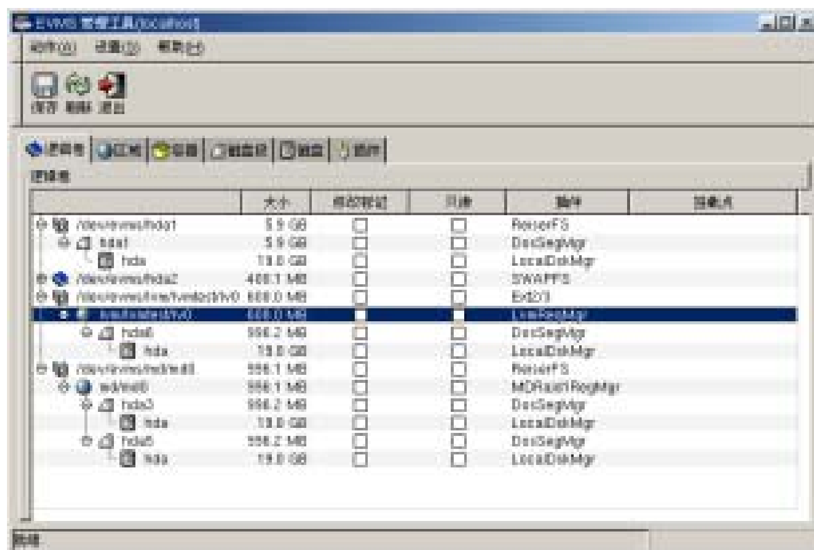


图5-1　　evmsgui 配置主界面

在配置主界面中，包括了 “**逻辑卷**”、“**区域**”、“**容器**”、“**磁盘段**”、“**磁盘**” 和 “**插件**” 等几个标签页。

点击不同的标签页，将显示出相应的系统 “总览图”，从而可以确切地知道从文件系统一直到底层保存数据的物理磁盘是如何分层的。

在 EVMS GUI 中，可以使用以下两种方式中的任何一种完成大部分任务：快捷菜单和动作菜单。

选择将要操作的对象，点击鼠标右键，就可以看到该对象的快捷菜单。从快捷菜单中选择将要执行的操作和配置任务即可。

作为一个管理员需要执行的所有操作都可以通过动作菜单获得。要使用动作菜单，在菜单栏中选择 “**动作**” 菜单中的各个子菜单项。动作菜单提供了一个比快捷菜单更具有可引导性的方式来完成任务。

5.3.2　使用 EVMS GUI 的例子

以下内容说明使用 EVMS GUI 如何完成一些最常见的任务，如：创建段、创建容器、创建区域、创建卷以及提交更改等。

5.3.2.1　创建一个段

如果要创建一个段，步骤如下：

1、 在菜单中选择 “**动作→创建→磁盘段**”，出现一个磁盘段管理器插件列表；

2、 选择 DOS Segment Manager（DOS 段管理器），单击 “**下一步**” 继续；

3、 下一个对话框中列出了适合于创建一个新段的可用空间存储对象。选择后单击 “**下一步**” 进入配置选项设置界面；

4、 此步骤为所选的可用空间对象进行配置选项设置，请输入或选择适当的值。标有 “*” 的域为必选项。DOS 段管理器提供了缺省值，但可以对其中一些值进行更改。一旦将信息填写到所有必需的域中，就可以按下 “**创建**” 按钮。



将鼠标指针放在某一个配置选项字段上，可以获得更多有关此选项的信息。

5、 单击 “**创建**” 按钮后出现一个结果显示窗口，单击 “**确定**” 即可。

成功创建了一个段之后，如果有可用的空余空间，还可以创建另外的磁盘段。

5.3.2.2　创建一个容器

如果要创建一个容器，步骤如下：

1、 选择菜单中的 “**动作→创建→容器**”，出现一个可支持容器创建的区域管理器插件列表；

2、 选择 LVM 区域管理器。单击 “**下一步**” 继续；

3、 下一个对话框中列出了适合的存储对象及其大小。LVM 区域管理器可以创建一个容器，选择早期创建的一个或多个段后，单击 “**下一步**”；

4、 在 “**配置选项**” 界面中，输入新建容器的名称及需要的物理单元大小。单击 “**创建**” 按钮。出现一个结果显示窗口，按 “**确定**” 即可。

一旦已经成功的创建了一个容器，如果有可用的空闲空间，还可以创建另外的容器。

5.3.2.3　　创建一个区域

如果要创建一个区域，步骤如下：

1、 选择菜单中的 “**动作→创建→区域**”，弹出一个区域管理器插件列表；

2、 选择 LVM 区域管理器，单击 “**下一步**” 继续；



在创建存储容器时，可以看见其它的选择列表之外的区域管理器。因为并不是所有的区域管理器都必须支持容器，所以可能无法看见所有区域管理器。

3、 从已创建的容器中选择可用的空闲空间区域，该容器应有一个类似于 lvm/yourvg/Freespace 的名称。单击 “**下一步**”；

4、 此步骤进行配置选项的设置。请输入或选择适当的值，标有 “*” 的域为必选项。在 “**新 LVM 区域的名称**” 中填写名称，根据需要修改其它域值，然后单击 “**创建**” 按钮；

5、 接下来弹出结果显示窗口，按 “**确定**” 按钮结束操作。

5.3.2.4　　创建一个卷

通过菜单中的 “**动作→创建→EVMS 逻辑卷**” 可以创建卷。为了示例如何使用快捷菜单，本节以快捷菜单操作说明创建一个卷的过程。

1、 点击 “**区域**” 标签页，查看该区域；

2、 用鼠标右键单击新创建的区域，在快捷菜单中选择 “**创建 EVMS 逻辑卷**”；

3、 在弹出的 “**创建 EVMS 逻辑卷**” 窗口中，键入该卷的名称，单击 “**创建**” 按钮。

单击 “**逻辑卷**” 标签页，列表中将显示新创建的逻辑卷。

5.3.2.5　　创建一个兼容性卷

上一节中创建的卷是一个 EVMS 逻辑卷，因此，它包含序列号和从设备号等 EVMS 详细信息。一旦使用这样的卷信息，该卷将不再完全与现有的卷类型向后兼容。

在前面的例子中，我们创建了一个磁盘分区（段）、一个 LVM 卷组（容器）和一个 LVM 卷（区域）。我们可以设置 EVMS，使其直接作为一个卷来使用，而不把 EVMS 系统数据添加到该区域。新卷称为一个兼容性卷。其它区域管理器中的段或区域可以使用相同的程序。使用此方法，最终产品将与要求的系统完全向后兼容。



如果要使上一节中创建的逻辑卷成为一个兼容性卷，必须先从存储对象中删除该卷。

如果要创建一个兼容性卷，用鼠标右键单击 LVM 区域，选择快捷菜单中的 “**创建兼容性逻辑卷**”，单击 “**创建**” 按钮。

单击配置主界面中的 “**逻辑卷**” 标签页，将出现一个与以上创建的区域具有相同名称的卷。这就是新

建的兼容性卷。

5.3.2.6　　保存更改

在提交更改前，EVMS GUI 中所做的所有更改只是储存在内存中。为了永久保存更改，在退出前必须保存所有更改。如果忘记保存更改而决定退出或关闭 EVMS GUI，它将提醒您执行保存操作。

如果要明确地对创建的文件提交更改，请在菜单中选择 “**动作→保存**”，并单击 “**保存**” 按钮。

5.4　EVMS Ncurses 接口

EVMS Ncurses（evmsn）用户界面提供与 EVMS GUI 具有类似特征的菜单驱动接口。像 EVMS GUI 一样，evmsn 可以在不需要改变任何代码的情况下提供新的插件和特征。

5.4.1　通过 EVMS Ncurses 浏览

在终端控制台的 shell 提示符下输入 evmsn 命令，启动 EVMS Ncurses 用户界面。参见下图 5-2 所示。

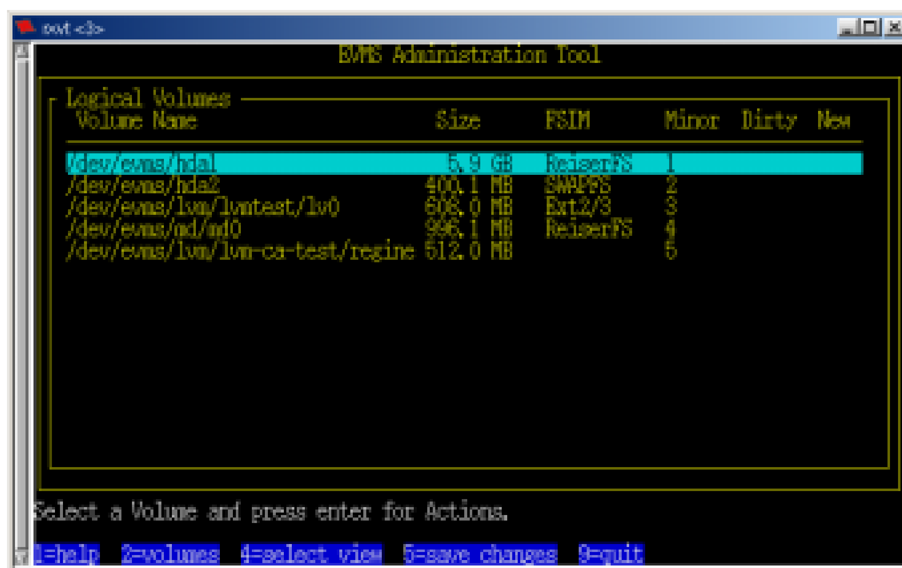


图5-2　　EVMS Ncurses 用户界面

可以看到，界面中显示了类似于 EVMS GUI 中的逻辑卷视图的逻辑卷列表，其中列出了当前系统中已经存在的卷。使用上下方向键可以在各个设备节点间移动。

界面底部列出了相关操作的功能键，具体如下：

<F1> 键或 **<1>** 键：显示帮助信息。

<F2> 键或 **<2>**键：返回卷视图。

<**F4**> 键或 <**4**> 键：显示可以切换的其它视图菜单。

<**F5**> 键或 <**5**> 键：保存在 evmsn 中所做的更改。

<**F9**> 键或 <**9**> 键：退出。

状态信息和用户提示信息显示在列表窗口与操作功能键提示行之间。

使用向上箭头和向下箭头选择要执行的动作，一旦该选项被突出显示，请按回车键。Ncurses 将提供相关可执行动作的子菜单。

当从一个菜单浏览到另一菜单时，使用 <**ESC**> 键可以返回到前面的菜单。



为了在配置选项菜单中改变或设置一个选项值，请使用空格键选择该值。特别注意的是，必须再一次按空格键获得输入新值的提示。完成操作后，请按回车。

5.4.2　使用 Ncurses 接口的例子

以下内容阐述如何使用 EVMS Ncurses 完成一些最普通的任务。这些任务包括创建段、创建容器、创建区域、创建卷和提交更改。

5.4.2.1　创建一个段

如果要创建一个段，步骤如下：

1、 按 <**F4**> 或 <**4**> 键切换视图；

2、 使用向下箭头向下滚动，突出显示 “**Disk Segments**” 项，按回车键继续；

3、 再次按回车键显示一个当前可选操作的子菜单；

4、 选择 “**Create a New Segment**”，按回车键，出现一个磁盘段管理器插件列表；

5、 选择 “**DosSegMgr（Dos Segment Manager）**”，按回车键继续；下一个对话窗口将列出适合创建一个新段的可用空间存储对象；

6、 移动上下箭头键选择合适的对象，然后使用空格键将对象选中。一旦选中了存储对象（用 X 标记出来），按回车键；

7、 接下来为所选的可用空间对象进行配置选项的设置。标有 “*” 的域为必需项。DOS 段管理器提供了缺省值，也可以根据需要改变其中一些值。

8、 突出显示某个域并按空格键，出现域值设置窗口。在 “**::**” 后输入新值以改变域值，完成所有的设置后按下回车键，新建的磁盘段将显示在段列表中。

成功创建了一个段之后，如果有可用的空余空间，还可以创建另外的磁盘段。

5.4.2.2　创建一个容器

如果要创建一个容器，步骤如下：

1、 按 <**F4**> 或 <**4**> 键切换视图；

2、 使用向下箭头键突出显示 "**Storage Containers**" 项，按回车键进入存储窗口列表；

3、 再次按回车键显示一个当前可选操作的子菜单；

4、 选择 "**Create a New Container**"，按回车键，出现一个可支持容器创建的区域管理器插件列表；

5、 选择 "**LvmRegMgr（LVM Region Manager）**"，按回车键继续。下一个窗口列出了适合的存储对象，如 LVM 区域管理器发现的适合用于容器创建的段、磁盘或区域；

6、 使用空格键选择列表中已经创建的一个或多个段，按回车键继续；

7、 按空格键选择容器名称域，在 "**::**" 提示符后键入新建容器的名称，然后按回车键；

8、 提示信息显示新的容器已成功创建，请按回车键完成操作。

5.4.2.3 创建一个区域

如果要创建一个区域，步骤如下：

1、 按 <**F4**> 或 <**4**> 键切换视图；

2、 使用向下箭头键突出显示 "**Storage Regions**" 项，按回车键进入存储区域列表；

3、 再次按回车键显示一个当前可选操作的子菜单；

4、 选择 "**Create a New Region**"，按回车键，出现一个区域管理器插件列表；

5、 选择 "**LvmRegMgr（LVM Region Manager）**"，按回车键继续；



在创建存储容器时，可以看见其它的选择列表之外的区域管理器。因为并不是所有的区域管理器都必须支持容器，所以可能无法看见所有区域管理器。

6、 从已创建的容器中选择可用的空闲空间区域。该容器应有一个类似于 lvm/yourvg/Freespace 的名称。使用空格键将其选中，按回车键继续，出现配置选项子菜单；

7、 输入新建 LVM 区域的名称，完成所需的其它更改，按回车键完成操作。

5.4.2.4 创建一个卷

如果要创建一个卷，步骤如下：

1、 在存储区域视图中突出显示新创建的区域，按回车键；

2、 在弹出的子菜单中选择 "**Create EVMS Volume from the Region**"，按回车键；

3、 确认将创建一个新卷，在 "**::**" 提示符后输入卷名，按回车键完成操作；

4、 按 <**F2**> 或 <**2**> 键查看卷视图，列表中将显示新创建的逻辑卷。

5.4.2.5　　创建一个兼容性卷

上一节中创建的卷是一个 EVMS 逻辑卷，因此，它包含序列号和从设备号等 EVMS 详细信息。一旦使用这样的卷信息，该卷将不再完全与现有的卷类型向后兼容。

在前面的例子中，我们创建了一个磁盘分区（段）、一个 LVM 卷组（容器）和一个 LVM 卷（区域）。我们可以设置 EVMS，使其直接作为一个卷来使用，而不把 EVMS 系统数据添加到该区域。新卷称为一个兼容性卷。其它区域管理器中的段或区域可以使用相同的程序。使用此方法，最终产品将与要求的系统完全向后兼容。



如果要使上一节中创建的逻辑卷成为一个兼容性卷，必须先从存储对象中删除该卷。

如果要创建一个兼容性卷，步骤如下：

1、 在存储区域视图中突出显示新创建的区域，按回车键；

2、 在弹出的子菜单中选择“**Create Compatibility Volume from the Region**”，按回车键；

3、 确认将创建一个新卷。由于卷名是自动分配的，所以不必提供卷名，按回车键完成操作；

4、 按 **<F2>** 或 **<2>** 键切换到卷视图，列表中将出现新创建的兼容性逻辑卷，该卷与前面创建的区域拥有相同的名称。

5.4.2.6　　提交更改

在提交更改前，EVMS GUI 中所做的所有更改只是储存在内存中。为了永久保存更改，在退出前必须保存所有更改。如果忘记保存更改而决定退出或关闭 EVMS GUI，它将提醒您执行保存操作。

如果要明确地对创建的文件提交更改，请按 <**F5**> 或 <**5**> 键并确认要保存更改。

5.5　EVMS 命令行解释器

EVMS 命令行解释器（EVMS CLI）为 EVMS 提供了命令驱动用户接口。EVMS CLI 设计为帮助实现自动化卷管理任务，在 EVMS GUI 无法使用的情况下，提供一个交互模式。

由于 EVMS CLI 是一个解释器，因此它的操作不同于操作系统中的命令行应用程序。对于 EVMS CLI 而言，只能在调用 EVMS CLI 的命令行中指定选项。这些选项控制 EVMS CLI 的操作方法，如：EVMS CLI 获取命令并进行解释的路径、EVMS CLI 对磁盘做出更改的提交频率、以及 EVMS CLI 报告其动作的长度。EVMS CLI 选项不是卷管理命令，只能在调用解释器时对其进行指定。 除非通过 -f filename 选项来调用 EVMS CLI，否则一旦调用，EVMS CLI 将对命令产生提示。如果通过 –f 选项调用 CLI，它将从指定的文件中获得命令。

在附有 EVMS 包的 grammar.ps 文件中指定 EVMS CLI 理解的卷管理命令。在 EVMS 手册页中对这些命令有详细描述，而且 EVMS CLI 本身就有这些命令的帮助信息。

5.5.1　使用 EVMS CLI 完成任务

使用 **evms** 命令启动 EVMS CLI，如果没有输入任何选项，EVMS CLI 将以交互模式启动。在交互模式下，EVMS CLI 产生对命令的提示。每个命令的结果将立即保存到磁盘。当输入一个 **exit** 时 EVMS

CLI 即退出。可以通过以下 **evms** 的选项对该动作行为进行修改。

-c　　在 EVMS CLI 退出时而不是在每个命令执行之后就对磁盘提交更改。

-f filename　　该选项指示 EVMS CLI 使用 filename 作为命令源。当 filename 的末端时，EVMS CLI 将退出。

-p　　该选项仅解析命令，并不真正执行。与 –f 选项结合时，常常用 -p 选项检测命令文件中的语法错误。

-h　　该选项和 **evms** 命令一起使用，显示选项帮助信息。



其余的选项通常很少使用。如有需要，可以在 EVMS 手册页中查找。

一旦已经调用 EVMS CLI，它将处理您提供的命令或命令文件。EVMS 手册页中提供了这些命令的完整列表。

以下部分检查一些普通的任务和可以用来完成这些任务的命令。

5.5.1.1　给新磁盘分配段管理器

当一个新磁盘添加到系统时，该磁盘常常没有被分区也不包含数据。最初，因为 EVMS 无法得知磁盘是否将作为一个卷使用，因此该磁盘在 EVMS 中以一个兼容性卷的形式出现。作为卷使用的磁盘与硬件 RAID 相同。硬件 RAID 将多个物理磁盘组合成一个虚磁盘，虚磁盘被分成区，每个分区在剩余系统中作为物理磁盘出现，这些磁盘则作为卷使用。

系统显示作为卷使用的物理磁盘。必须告知 EVMS 新磁盘不能作为卷使用。使用 "**Revert**" 命令完成这项任务。以下示例说明该过程。

假设在被 EVMS 当作 sde 的系统中添加一个新磁盘，该磁盘不含有数据并且没有以任何方式进行划分（没有分区）。EVMS 假设该磁盘是一个称作 /dev/evms/sde 的兼容性卷。告知 EVMS 该磁盘不是一个卷，并可以通过输入以下命令来使用：

Revert：/dev/evms/sde

一旦 EVMS 得知该磁盘可以使用，可以给该磁盘分配一个段管理器。该段管理器允许将该磁盘分段（分区）。以下示例说明该过程。

键入以下命令，给该磁盘和 sde 分配 DOS 段管理器。

Assign：DosSegMgr={},sde

以上命令将 DOS 段管理器分配给 sde。DOS 段管理器将在磁盘上创建两个段：一个被称为 sde_mbr 的元数据段和一个代表驱动器中可用空间的段 sde_freespace1。由于该可用空间段（sde_freespace1）在没有使用的驱动中表现空间，所以它可以分为其它段。

5.5.1.2　创建一个段

数据段代表用于构建卷的磁盘空间。元数据段代表磁盘的存储空间，在该磁盘中管理磁盘的段管理

器消耗其空间。可用空间段表现未用的磁盘空间。通过在可用空间段中将一部分或所有的可用空间分配给数据段，将可用空间段创建为数据段。无论何时分配给磁盘的段管理器需要更多存储空间，该段管理器都将可用空间段创建为元数据段。

如果要将一个可用空间段创建为一个数据段，请使用 “**Allocate**” 命令，示例如下：

将可用空间段 sde_freespace1 创建为一个 100 MB 的段，并置于由 DOS 段管理器控制的驱动器中。

使用 “**Allocate**” 命令完成以上例子。“**Allocate**” 命令获得以下变量：可用空间段名称是由段管理器分配的，并且该段管理器是从选项列表中获得的。在我们的例子中，DOS 段管理器使用大小选项指定数据段的大小，将可用空间段创建为数据段。完成该项任务的命令如下：

Allocate：sde_freespace1,size=100MB

也可以使用 “**Create**” 命令完成以上例子。“**Create**” 命令接受的变量因依靠创建的对象的不同而不同。“**Create**” 命令的第一个变量指出要创建的对象，即上面例子中的段。剩余的变量是由段管理器分配的可用空间段并且该段管理器是从选项列表中获得的。完成该项任务的命令如下：

Create：Segment,sde_freespace1, size=100MB

5.5.1.3　创建一个容器

段和磁盘结合在一起形成容器。容器允许组合存储对象，然后把这些组合的存储对象分为新的存储对象。可以结合存储对象执行如同在 AIX 和 Linux LVM 卷管理器中的卷组概念。

例如：一个系统有三个可用的磁盘驱动（sdc、sdd、hdc），使用 EVMS LVM 区域管理器将这些磁盘驱动合成 16 KB 的 PE 大小、称为 “Sample Container” 的容器。

“**Create**” 命令用于创建容器。在此例的容器中，“**Create**” 命令中的第一个变量是产生一个容器的对象类型。那么，“**Create**” 命令接受以下变量：区域管理器与其需要的任何参数、创建容器的段和磁盘一起使用。完成以上例子的命令是：

Create：Container,LvmRegMgr={name="Sample

Container",pe_size=16KB},sdc, sdd,

hdc

以上命令指定使用 LVM 区域管理器将 sdc、sdd、hdc 组合成一个容器。LVM 区域管理器支持名称和 pe_size 选项。这些选项为容器提供名称和 pe_ 大小。

5.5.1.4　创建一个区域

区域通常由容器创建而来，但也可以由段和磁盘创建。大多数支持容器的区域管理器将创建一个或多个可用空间区域来体现容器中的可用空间。该功能类似于段管理器创建可用空间段来表现未用磁盘空间的方法。使用 “**Allocate**” 命令，将这些可用空间区域创建为数据区域，示例如下：

给定一个名为 “Sample Container” 的容器，它有 8799 MB 的单独可用空间区域，用它来创建一个 1000 MB 的数据区域。

LVM Region Manager 支持创建区域的多个选项。如果要查看创建区域和容器的可用选项，请使用 “**Query**” 命令：

Query：plugins,plugin=LvmRegMgr,list options

在上例子中使用这些选项创建该区域。“**Allocate**” 命令获得以下参数：可用空间段是由段管理器分配的，该段管理器是从选项列表中获得的并控制分配的可用空间区域。需要的 LVM 区域管理器选项是名称和大小。也可以使用其它选项，但这些选项是必需的。如果要完成该例子，请键入以下命令：

Allocate："lvm/Sample Container/Freespace",name="Sample Region",size=1000MB

组合段、磁盘或区域创建的区域不能使用 “**Allocate**” 命令。这些区域使用 “**Create**” 命令。适用于 “**Create**” 命令的变量如下：关键字区域、区域管理器使用的名称、区域管理器选项以及组合形成新区域的段、磁盘和区域。该命令的形式是：

Create ： region, Region Manager Name = {Region Manager Options} , Segment/Disk/Region,Segment/Disk/Region …

5.5.1.5　　创建一个存储对象

从段、区域、磁盘或其它存储对象创建存储对象。当希望在一个卷中获得一个 EVMS 的特殊特征，如不良（存储）块重定位（BBR），主要使用存储对象。使用 “**Create**” 命令创建存储对象。在这种情况下，适用于 “**Create**” 命令的变量是关键字 “**对象**”，该对象具有与其选项一起使用的 EVMS 特征的名称以及用于创建新存储对象的段、磁盘、区域和存储对象。

例如： lvm/Sample Container/Sample Region 区域，使用 EVMS 的不良块重定位特征，创建一个名为 BBR_Region 的存储区域。

由于 BBR 特征仅支持一个选项和名称，完成该项任务的方法如下：

Create：Object,BBR={name=BBR_Region},“lvm/Sample Container/Sample Region”



EVMS CLI 忽略空间。所有包含空间的名称必须用引号括起来。

5.5.1.6　　创建一个卷

从磁盘、段、区域或存储对象创建卷。卷是 EVMS 本地卷或兼容性卷。除了 EVMS，兼容性卷还与一个卷管理器相兼容，如 Linux LVM。在 EVMS 操作兼容性卷的用途方面，兼容性卷可能存在局限性。EVMS 本地卷没有这种局限性，但只有在 EVMS 装配系统中才能使用。

- **本地卷举例**

从名称为 BBR_Region 的区域创建为一个 EVMS 本地卷，该区域是在前面的例子中创建的。

完成以上例子的命令是：

Create：Volume, BBR_Region, Name = "Sample EVMS Native Volume"

- **兼容性卷举例**

假如一个由 LVM 区域管理器创建的区域，该区域管理器命名为 lvm/Sample Container/My LVM Volume，并且没有成为其它任何区域或存储对象的一部分。创建一个与 Linux LVM 兼容的卷。

因为 LVM 区域管理器创建 lvm/Sample Container/My LVM Volume 区域，且该区域对 LVM 区域管理器没有做任何操作，所以，该区域是 Linux LVM 能识别的格式。键入以下命令，将该卷变为一个兼容性卷：

Create：Volume,"lvm/Sample Container/My LVM Volume",compatibility

5.5.2　命令及命令文件的注意

EVMS CLI 允许多个命令出现在一个命令行中。当在一个命令行中使用多个命令时，这些命令必须用冒号隔开，这就是命令分隔符。因为 EVMS CLI 将一个命令文件看作一个单独的长命令行，所以这对于命令文件是十分重要的。EVMS CLI 在文件中没有行的概念，而且忽视空间。这些功能允许命令行中的一个命令跨越几行并使用任何一种缩进或易于使用的边距。唯一的要求就是命令分隔符出现在两个命令之间。

除非引号内出现空间，否则 EVMS CLI 忽视空间。所有包含空间或其它不可印刷或控制字符的名称都应放在引号内。如果名称中的引号是该名称的一部分，则该引号必须是 “双引号”，如下所示：

"This is a name containing ""embedded"" quote marks."

EVMS CLI 关键字不区分大小写，但 EVMS 名称有大小写之分。

最后，EVMS CLI 支持 C 编程语言风格注释。除了不能在一个引用行内开始和结束外，注释可以在任何一处开始和结束，如下所示：

/* This is a comment */

Create：Vo/*This is a silly place for a comment, but it is

allowed.*/lume,"lvm/Sample Container/My LVM

Volume",compatibility

5.6　LVM 应用

Linux LVM 以卷组概念为基础。卷组（VG）是物理卷（PV）的集合。一个组中所有的 PV 都有其自己的存储空间，可以再细分为多个称为物理范围（PE）的小的、有固定大小的空间。一个 PE 的默认大小是 4 MB。通过将一个或多个 PE 分配并创建逻辑卷（LV）。当 I/O 请求对一个卷发出时，由 LVM 代码确定将该请求放在哪一个 PV 和 PE 中，并将该请求传递到合适设备的堆栈中。

如果您很熟悉 Linux LVM，并且希望在 EVMS 下使用现有的 LVM 卷。这种情况下，您可能已经熟悉了 LVM 命令设置，熟悉使用命令来执行一些特殊任务，如：创建一个卷组或创建一个逻辑卷。

除了实现和管理 LVM 卷的 EVMS 核心和用户空间插件之外，使用 EVMS Engine APIs 效仿 LVM 命令设置写了一套命令行应用程序。

目前，在命令行应用程序中可以使用以下命令。一般来说，选项与那些可用的 LVM 命令是相同的。对每个命令使用 “--help” 选项，可以了解更多有关该命令的信息。

evms_vgcreate　通过 PV 的给定列表，创建一个新的 LVM 卷组。VG 命令不再要求用户使用 evms_pvcreate 准备对象。目前，这些命令可以直接使用 EVMS 中的所有可用对象。同样，evms_pvremove 命令不再需要释放 PV 返回到可用的 EVMS 对象中。

evms_vgremove　删除一个现有的卷组。

evms_vgextend　在现有的卷组中添加新 PV。

evms_vgreduce　从现有的卷组之外获得 PV。

evms_lvcreate　创建一个新的 LVM 逻辑卷。**evms_lvcreate** 支持创建标准（线性）LV 以及条带 LV。线性 LV 可以指定为相邻 LV。LVM 插件定义了相似含义，即在 PE 的相邻运行中，LV 必须存在于一个单独的 PV 中。该命令也支持在相同 VG 中创建已有 LV 的快照，并允许为所有非快照 LV 指定基本 PV。

evms_lvremove　删除一个 LVM 逻辑卷。不允许删除快照 LV。

evms_lvextend　通过增加范围，扩大一个已有的 LV。扩大 LV 之后，必须扩大文件系统。目前，该命令不支持扩大快照或原始快照。

evms_lvreduce　通过删除范围，缩小一个已有的 LV。缩小 LV 之后，必须缩小文件系统。不要首先缩小 LV，否则将有从文件系统中丢失数据的危险。目前，该命令不支持缩小快照或原始快照。

evms_pvscan　列出当前所有的 LVM PV。该命令也同样列出其它所有的 EVMS 磁盘、段、区域和标记，如：“available” 或 “unavailable”。可用对象可以直接用来创建或扩展卷组。

evms_vgscan　列出所有已有的 LVM VG。该命令也包括一个管理提交的新选项（-c）和创建在 /dev 下所有必需设备文件。该文件的创建对于已有的 LVM 的安装是十分必要的，该安装没有使用 EVMS 工具创建任何新的 LV（除非启用 devfs）。使用 EVMS GUI 和执行提交可以得到相同的结果。

evms_lvscan　列出所有已有的 LVM LV。当一个 LV 列为不活动的，LV 在 EVMS 中不显示为一个兼容性卷，并且不能通过 EVMS 来使用。可以使其出现的唯一方法是在 EVMS GUI 中创建 LV 并且不添加一个兼容性卷。使用 GUI 执行给一个不活动的 LV 添加一个兼容性卷的任务。

evms_pvdisplay　显示与指定的 PV 相关的其它信息。

evms_vgdisplay　显示与指定的 VG 相关的其它信息。

evms_lvdisplay　显示与指定的 LV 相关的其它信息。

两个命令已经从 LVM 应用程序中删除。不再需要 **evms_pvcreate** 和 **evms_pvremove**。**evms_pvscan** 命令此时显示所有 LVM PV，以及 EVMS 中的所有其它磁盘、段和区域，并且将其标记为可用或不可用。**evms_vgcreate** 可以使用标记为可用的对象而不需要调用 **evms_pvcreate**。

前面已经讲过，EVMS 使用的术语与 LVM 使用的术语不同。但是，两套术语是互相对等的。LVM 中的卷组（VG）相当于 EVMS 中的容器。因为 LVM 插件是一个 EVMS 区域管理器，所以它的输出对象（LVM 中的 LV）被称作区域。即使区域管理器确实能够获得磁盘、其它区域和输入，输入对象（LVM

中的 PV）还是被称为段。命令行应用程序经常交换使用 LVM 和 EVMS 术语，所以希望不要混淆这些简单的规则。

上面列出的每个命令的选项应与 LVM 相关命令的选项一致。但是，现在一些选项的含义有细微的差别。例如，冗长选项（-v）不仅为用户显示其它与命令相关的信息，而且开放有 DEBUG 级信息的引擎，所以将其它信息写入引擎日志（/var/log/evmsEngine.log）。调试选项（-d）与冗长选项一样，为用户显示相同数量的信息，但是开放有 ENTRY_EXIT 级信息的引擎，这可以使非常详细的跟踪信息写入引擎日志。同时，由于一些选项的特征仍没有实施，或在 EVMS 中不需要这些特征，所以目前忽视了这些选项。例如，自动备份选项（-A），由于 LVM 插件仍没有将元数据备份，所以忽视了它。

5.7　EVMS 插件描述

下面的内容说明在 EVMS 中使用的插件。

5.7.1　不良块重定位特征插件

不良块是不能使用的磁盘扇区，这是因为访问扇区的所有尝试因 I/O 错误而导致的失败。不良块重定位（BBR）特征通过在同一磁盘上重新映射该扇区到其它扇区以检测并更正不良块。因为 BBR 使用磁盘块代替不良块，因此保留该映射的磁盘扇区被称为重定位块。一旦 BBR 将一个不良块重新映射到一个重定位块中，磁盘访问则不会产生 I/O 错误。

配置一个 BBR 存储对象时，该插件保留磁盘中的一个区域，以便保证提供重定位块。BBR 消耗重定位块的供给，因为 BBR 重映射不良块，直到没有更多的重定位块可用为止，因此。当没有更多的重定位块可用时，由于插件无法使用重定位块重映射不良块，所以，所有不良块的 I/O 都将失败。

通过选择一个已有的对象并为新的 BBR 存储对象提供一个名称，创建一个 BBR 存储对象。BBR 插件自动配置重定位块数量以及在磁盘中的位置。重定位块的数量应少于轨迹。当调整一个卷或存储对象的大小时，BBR 存储对象不会成为缩小点或扩大点，所以该插件没有调整大小的选项。

5.7.2　DOS 段管理器插件

DOS 段管理器插件是一个磁盘分区管理器，它负责发现并管理下列情况：

- MSDOS 磁盘分区
- 嵌入式 Unixware 扇区
- OS/2 分区和 Dlat 扇区
- 嵌入式 bsd 扇区
- 嵌入式 Solarisx86 扇区

磁盘分区是在磁盘上有起始地址和大小的区域。分区类型多种多样。分区信息保存在被称为分区记录的结构中，分区记录汇集在被称为分区表的结构中，分区表存储在磁盘上，所以固件、操作系统和磁盘分区工具都可以发现磁盘分区。

DOS 段管理器插件创建映射到磁盘分区的段存储对象。当创建、损坏或调整段存储对象时，DOS 段管理器重新构建磁盘分区表，在命令提交更改时反映该更改并将新表写入磁盘。DOS 段管理器更新

MSDOS 分区表、嵌入式分区表和 OS/2 dlat 扇区。

在创建一个段存储对象时，必须从可接受可用空间段列表中选择对象。可用空间段对象常用于表现磁盘上没有被分区或分区表使用的区域。因为段存储对象已经创建，可用空间内的可用扇区被分配到新的分区和分区表中。选择可用空间存储对象确定在磁盘中创建新分区的位置。选择一个可用空间段之后，需要根据下列配置进行选择：

大小	该值为扇区数量，必须在柱面大小的总数中。
偏移量	该值为扇区数量，该值允许偏移新分区的开始。
类型	该值为一个整数，确定创建分区的类型，默认到 Linux 0x83 中。
可引导	该值为 YES 或 NO，如果希望将分区标记为活动的，请选择 YES。
主要的	该值为 YES 或 NO，如果希望创建一个主分区，请选择 YES。

如果正在 OS/2 磁盘上创建一个段存储对象，有三种附加域：

分区名称	值是一个字符串，OS/2 分区有名称，最多 20 个字符。
卷名称	值是一个字符串，OS/2 逻辑卷有名称，最多 20 个字符。
驱动器号	值是类型字符，OS/2 兼容性卷有驱动器号（A-Z）。

在调整一个卷或存储对象的大小时，一个段存储对象也许成为缩小或扩展点。当缩小一个段存储对象时，仅有一个可接受的对象，并且该对象本身就是段对象。只有磁盘中有紧跟着该对象的未使用的扇区时，才可以扩展一个段存储对象。未使用的扇区被称为可用空间。系统中包括一个映射未使用扇区的可用空间段存储对象。当扩展一个段存储对象时，唯一可接受扩大的对象是可用存储对象，该对象映射紧跟着段存储对象的未使用的扇区。

为了调整段存储对象的大小，一旦选择可接受对象，就必须指定要扩展或缩小段对象的扇区的数量。因为磁盘分区位于柱面边缘，所以必须经常扩大或缩小柱面大小总数以保持最后柱面边缘请求。

5.7.3　驱动连接特征插件

驱动连接特征插件是一个聚合插件，它将几个较小的存储对象汇集在一起，产生一个存储对象。获得的聚合存储对象被称为驱动连接存储对象。被驱动连接对象消耗的较小对象称为驱动连接子对象。大多数 EVMS 存储对象可作为子对象使用，如磁盘、区域和分区。

一个驱动连接存储对象是子对象的有序集合。驱动连接插件维持该集合的顺序。当首先创建驱动连接存储对象时，挑选的子存储对象的顺序决定该集合的顺序。在每个子对象中，通过保存该信息的副本来保持此顺序。每次规则信息允许插件以相同顺序重新组合子对象。规则信息也确保每次磁盘访问正确映射到相同的扇区中。

可以通过为驱动连接选择子对象并为新的驱动连接存储对象提供一个名称，创建一个驱动连接存储对象。在调整一个卷或存储对象的大小时，一个驱动连接存储对象可能成为缩小或扩展点，可以通过选择扩展或缩小的对象并调整该对象的大小，调整卷或存储对象的大小。



只有在已有的驱动连接的末端添加或删除子对象，才能调整一个驱动连接存储对象的大小。该局限性防止任何对子对象的损坏。

如果要扩大卷或存储对象，请从可接受存储对象列表中选择几个对象，并且在已有的驱动连接存储对象的末端添加这些对象。当缩小一个卷或存储对象时，可以从驱动连接末端中选择删除几个存储对象。只有删除驱动连接中最后一个子对象，才可以缩小驱动连接存储对象。如果选择几个子对象来缩小一个驱动存储对象，那么所选对象的顺序十分重要。必须首先选择驱动连接中最后一个子对象。然后选择最后一个子对象的第二个子对象。

5.7.4　全局唯一标识符（GUID）分区表段管理器插件

全局唯一标识符（GUID）分区表段管理器插件是一个磁盘分区管理器，它负责发现并管理下列情况：

- GPT 磁盘分区
- GPT 分区磁盘上的可用空间
- GPT 保护主引导记录（PMBR）

磁盘分区是一个在磁盘上有起始地址和大小的区域。分区类型多种多样，包括系统分区、基本数据分区，以及一些保留的分区类型。分区信息保存在一个被称为分区记录的结构中，分区记录汇集在被称为分区表的结构中。在 GPT 磁盘上，分区表有两个副本。这些表存储在磁盘上，所以固件、操作系统和磁盘分区工具可以查找到磁盘分区。

GPT 段管理器插件创建映射到磁盘分区的段存储对象。当创建、损坏或调整段存储对象时，DOS 段管理器重新构建磁盘分区表，命令提交更改之后反映该更改并将新表写入磁盘。GPT 段管理器更新 GPT 标题表和 GPT 分区表。

创建一个段存储对象时，必须从可接受可用空间段列表中选择对象。可用空间段对象常用于表现磁盘上没被分区或分区表使用的区域。因为段存储对象已创建，所以可用空间内的可用扇区被分配到新的分区和分区表中。选择一个可用空间存储对象确定在磁盘中创建新分区位置。选择一个可用空间段之后，必须做出基于以下配置的选择：

大小	值是扇区数量，必须在柱面大小总数中。
偏移量	值是扇区数量，该值允许您偏移新分割区的开始。
类型	目前仅允许基本数据分区。

如果调整卷缩小或扩展的点是一个段存储对象，那么段存储对象是可接受的调整象。如果磁盘上有紧跟着的未使用扇区，就只能扩展段存储对象。当扩展一个段存储对象时，唯一可接受的扩展对象是可用存储对象，该对象映射紧跟着段存储对象的未使用的扇区。

为了调整段存储对象的大小，一旦选择了可接受对象，就必须指定要扩展或缩小段对象所使用的扇区的数量。因为磁盘分区位于柱面边缘，所以必须经常扩展或缩小柱面大小总数以保持最后柱面边缘请求。

5.7.5　Multi-Disk 插件

EVMS Multi-Disk（MD）插件为 Linux software RAID 卷提供支持。EVMS MD 插件读取由本地 Linux MD 驱动器写入的磁盘，并用相同的格式将元数据写入新的 RAID 设备中，所以本地 Linux MD 驱动器可以对其进行读取。

EVMS MD 插件为线性、RAID0、RAID1、RAID4 和 MD 的 RAID5 个性化提供支持。

- 线性支持不断与许多设备连接并使它们以一个巨大的设备出现。
- RAID0 跨越许多设备提供条带化。
- RAID1 提供 n-way 镜像。n-way 镜像是指每个设备都包括一个同一数据的副本。
- RAID4 和 RAID5 提供奇偶条带化。即使 RAID 中的设备出现故障，奇偶也可以重新找回数据。RAID4 在一个设备中保留了所有奇偶。RAID5 使用四种不同算法之一，在各设备间分配奇偶。

当 RAID 运行时，EVMS MD 允许在 RAID1、RAID4 和 RAID5 中添加或删除空闲磁盘。当 RAID 运行时，该插件允许在 RAID0 n-way 镜像中添加或删除设备。在系统运行时，只要有空闲设备代替删除的设备，该插件也允许从 RAID4 或 RAID5 阵列中删除设备。

如果选择使用 EVMS MD 插件，就不能同时使用 Linux 本地 MD 驱动器。如果两个设备同时管理 MD 设备，这将导致数据损坏。

5.7.6　快照特征插件

快照是卷的 “冻结” 图像。也就是说，在特定的情况下及时获得一个卷的图像，并且在安装和修改原始卷时允许访问该静止状态。该静态图像可以以不同的卷进行安装，并且在这种过程中使用，如备份等。

除了静态功能，EVMS 快照插件允许将快照创建为读取/写入。实际上，该附加部分是原始卷的一部分。在不影响原始卷的情况下，新卷是原始卷的一个可以写入的虚拟副本。快照可以在所有 EVMS 卷中创建。应注意的是，除非将快照对象转换成一个卷，否则快照不会真正启动。同时，回复快照卷禁用了快照并使其重新设置。该动作对于定期进行快照和备份的卷是十分有用的。

在 EVMS 1.1.0 中，快照插件支持 “回滚” 特征。该特征利用在一些点及时获得卷的快照代替该卷。用卷快照代替该卷的特征允许将一个卷返回到获得快照时的状态。在获得的快照后，原始卷的更改消失。如果该快照是可写入的，那么写入快照的更改成为 “回滚” 卷的一部分。为了执行回滚，不能安装原始卷和快照。

EVMS 快照插件也支持异步模式。异步模式允许更快的快照。异步快照通过正常关闭保存；但是，它们不能在系统悬挂或电源切断时保存。

5.8　文件系统接口模块（FSIM）

目前，EVMS 装载了 3 个文件系统接口模块（FSIM）。这些模块允许 EVMS 与文件系统应用程序相交互，确保 EVMS 安全地执行操作，如：通过和文件系统协调动作进行扩大和缩小。如 **mkfs** 和 **fsck** 的操作也可以在其它 EVMS 用户界面中执行。在用户界面中保存更改之后，所有通过 FSIM 发起的动作都向磁盘提交。

5.8.1　JFS

JFS 模块支持 IBM 日志文件系统（JFS）。现有支持包括 **mkfs** 和 **fsck**。在不久的将来，将添加对外部日志和扩展的支持。要使用 EVMS FSIM，系统至少需要有 JFS 应用程序的 1.0.9 版本。最新的应用程序可以从 http://oss.software.ibm.com/jfs 网站下载。

5.8.2　ReiserFS

ReiserFS 模块支持 ReiserFS 日志文件系统。除了 **mkfs** 和 **fsck**，该模块也支持文件系统的脱机调整（扩展和缩小）。要使用 EVMS FSIM 模块，必须有 ReiserFS 应用程序 3.x.1a 或更高版本。ReiserFS 应用程序的最新版本可以从 http://www.namesys.com/ 网站下载。

5.8.3　EXT2

EXT2 模块既支持 ext2 文件系统格式，也支持 ext3 系统文件格式。它支持 **mkfs** 和 **fsck**，也支持脱机扩展和缩小。EVMS EXT2 模块要求 e2fsprogs 包的 1.20 或更高版本。e2fsprogs 包的最新版本可以从 http://e2fsprogs.sourceforge.net/ 网站下载。

5.8.4　SWAPFS

SWAPFS FSIM 支持 Linux 开关设备，它允许创建并删除开关设备。目前，用启动脚本或手动的方式发出 **swapon** 和 **swapoff** 命令。只要开关设备没有使用，该 FSIM 就可以调整开关设备。

第 6 章　内核升级工具

新版本的内核会修正旧版内核中的一些错误和缺陷，提供更多的设备驱动程序和其它一些有用的新特性。所以通过升级内核往往可以获得更好的硬件兼容性、更高的效率和稳定性。

Red Flag Linux 使用 RPM 软件包格式构建内核，这样的格式易于升级和管理。而且，Red Flag Advanced Server 4.0 中提供了一个升级内核的图形化工具——rfupdatekernel，使升级内核的工作变得更为容易。

本章介绍在 Red Flag Advanced Server 4.0 系统中使用 updatekernel 工具升级内核的必要步骤。

6.1　准备工作

- **了解现有的内核版本号**

用 uname –a 命令来查看系统当前的核心版本号。如：

```
# uname –a
Linux localhost 2.4.21-AS.2smp #1 SMP 一 8 月 11 20:25:51 CST 2003 i686 i686 i386 GNU/Linux
```

表示当前系统的内核版本是 2.4.21-AS.2smp。

- **获取新内核**

请选择由红旗公司构建发布的内核软件包（它们是经过严格测试的），本工具不保证非红旗公司制作的内核包可以被正确升级。

- **了解新内核的基本信息**

请参阅随内核软件包发布的文档，了解新内核的基本信息。

- **创建引导盘**

虽然内核升级工具会在安装新内核的同时保留系统中原有的内核及其 LILO 引导项，但我们仍然建议用户创建一张可运行的紧急引导盘。万一发生不能正确引导系统的等特殊情况，可以使用这张引导盘启动和恢复系统。

要创建一张可以使用当前运行的内核来引导系统的引导盘，执行以下命令：

```
# mkboootdisk ‘uname –r’
```

制作了引导盘后，请使用它来重新引导系统以检验它可以正确使用。

6.2　启动内核升级工具

需要进入 KDE 桌面环境，以 root 权限启动和运行内核升级工具。可以采用以下方法启动：

1、 在系统主菜单中选择“**系统→控制面板**”，打开控制面板，在“**系统配置**”标签页中，双击“**红旗内核升级工具**”；

2、 在系统主菜单中选择“**管理工具→红旗内核升级工具**”；

3、 在运行命令窗口或 shell 提示符下直接键入 rfupdatekernel 。

6.3　执行升级

准备工作完成，启动内核升级工具后，首先看到下图 6-1 所示的欢迎界面：



图6-1　　欢迎界面

单击“**下一步**”继续，进入“**选择内核升级方案**”界面，参见下图 6-2。

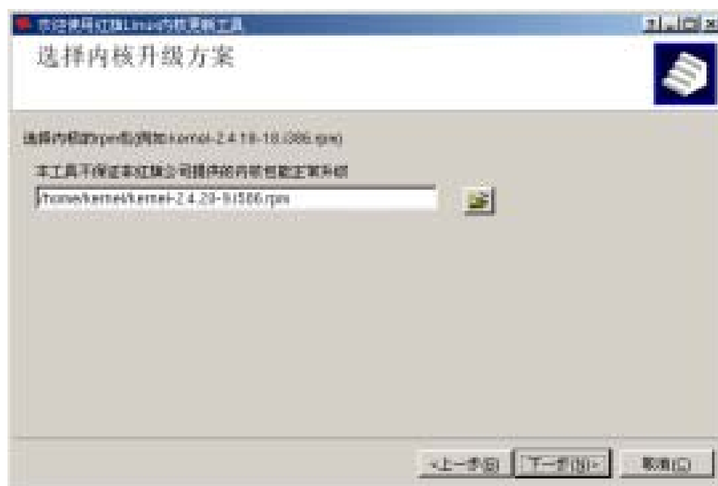


图6-2　　选择内核的 rpm 包

在文本框中输入将要升级的内核 RPM 包在系统中的位置，也可以点击 图标，在文件浏览窗口中选择内核的 RPM 包文件。



对于单处理器系统而言，只有 kernel 软件包是必需的。如果计算机中不只有一个处理器，则需要升级支持多处理器的 kernel-smp 软件包。

接着按 “**下一步**”，进入 “**用户选择信息**” 窗口。



图6-3　　提示信息

窗口中显示了刚才所选择的将升级内核软件包的信息，包括软件包的路径和版本号，请用户确认。确认无误后，请单击 “**下一步**”，正式开始升级内核的操作。

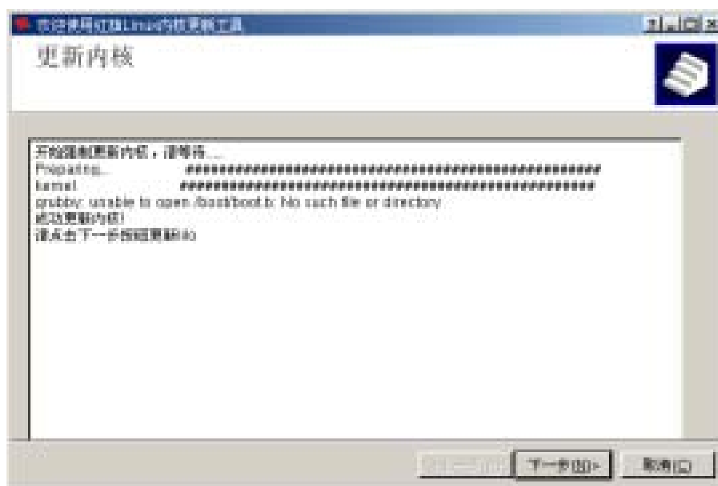


图6-4　　内核升级过程提示

此步骤一般需要几分钟的时间。系统首先进行软件包和内核升级的准备工作，然后将新的内核安装到系统中。

Red Flag Linux 的 kernel RPM 包会在安装过程中按照包配置文件中预定义的方式处理系统中的 GRUB 或 LILO 引导装载程序（Red Flag Server 4.0 系统使用 LILO 引导）。不过，为了使用户能够根据自己的需要灵活地定制 LILO 引导装载程序，单击 “**下一步**” 将进入设置 LILO 配置文件窗口。

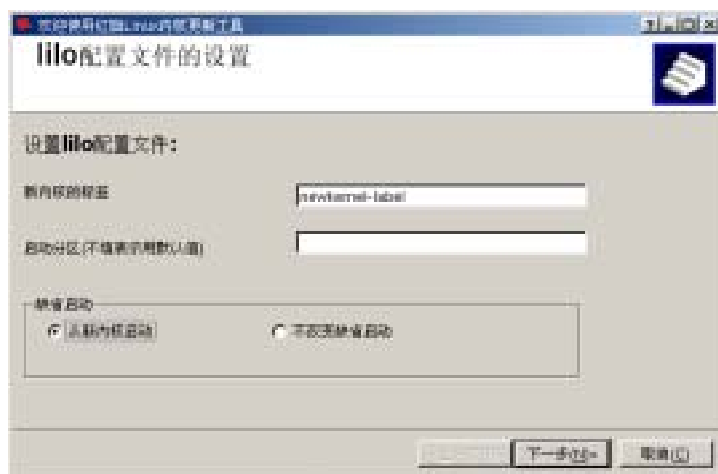


图6-5　　设置 lilo 配置文件

“**新内核的标签**”：输入新内核的引导卷标，即当 LILO 启动后，在菜单中显示的用于引导新内核的标识；***注意：卷标的长度不能超过 14 个字符。***

“**启动分区**”：一般不需要填写；

“**缺省启动**”：设置 LILO 的默认引导项，默认的选择是 “**从新内核启动**”，如果暂时不想使用新的内核，可以选择 “**不改变缺省启动**”。

设置完成后，单击 “**下一步**” 继续。



图6-6　　更新 lilo

图 6-6 所示的界面列出了更新 LILO 的具体步骤：首先修改 LILO 配置文件 /etc/lilo.conf，然后运行 /sbin/lilo 命令来启动改变。界面提示添加了新内核的引导且成功更新了 LILO！

单击 “**下一步**”，进入内核升级工具的最后一个界面，如图 6-7 所示。



图6-7　　完成内核的升级

完成内核的升级后，需要重新启动系统，请确认是否 “**立即重启系统**”，然后按 “**完成**” 按钮结束操

作。

6.4　LILO 配置文件示例

下面的例子说明了安装升级内核前后 LILO 配置文件 /etc/lilo.conf 的变化。

<table>
<tr><th>升级前</th><th>升级后</th></tr>
<tr><td><pre>prompt
timeout=50
default=Advanced
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
lba32

image=/boot/vmlinuz-2.4.21-AS.2
        label=Advanced
        initrd=/boot/initrd-2.4.21-AS.2.img
        read-only
        root=/dev/hda1
image=/boot/vmlinuz-2.4.20-8
        label=linux
        initrd=/boot/initrd-2.4.20-8.img
        read-only
        root=/dev/hda1</pre></td><td><pre>prompt
timeout=50
default=update
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
lba32

image=/boot/vmlinuz-2.4.22-2
        label=update
        initrd=/boot/initrd-2.4.22-2.img
        read-only
        root=/dev/hda1
image=/boot/vmlinuz-2.4.21-AS.2
        label=Advanced
        initrd=/boot/initrd-2.4.21-AS.2.img
        read-only
        root=/dev/hda1
image=/boot/vmlinuz-2.4.20-8
        label=linux
        initrd=/boot/initrd-2.4.20-8.img
        read-only
        root=/dev/hda1</pre></td></tr>
</table>

第 7 章　异步 I/O 指南

7.1　简介

异步 I/O 允许用户在一个 I/O 操作处于运行状态时，执行另一个有用工作。也就是说，在异步 I/O 中，请求发送给操作系统，同时下一条指令立即执行。因为 I/O 操作和应用程序进程是同时进行的，所以使用异步 I/O 会提升性能，而且通常会提高系统的 I/O 吞吐量。

Red Flag Advanced Server 4.0 支持异步 I/O，包括裸设备异步 I/O 和文件系统异步 I/O。使得数据库，文件服务器等应用可以利用异步 I/O，从而使得多个 I/O 操作同时执行，提高了系统的性能。

7.2　异步 I/O 子程序

以下子程序为执行异步 I/O 而提供：

aio

名字

aio—异步 IO。

摘要

include <errno.h>

include <aio.h>

描述

POSIX.1b 标准定义了一组新的 I/O 操作，此操作能够明显减少 I/O 的请求等待时间。当并行操作 I/O 时，该新功能允许使用程序进行一个或多个 I/O 操作并立即恢复正常工作。*unistd.h* 文件定义符号为 **_POSIX_ASYNCHRONOUS_IO** 时就会实现此功能。

这些功能是称为 librt libc 二元的具有实时功能库中的部分功能。通过使用内核程序（如果存在）中的支持或者使用用户级的基于线程的执行命令，可以执行这些功能的操作。在使用用户级的基于线程的执行命令中，除了与 librt 和 libaio 连接之外，还有必要将应用软件与线程库 libpthread 相连接。

AIO 的所有操作都是对以前打开的文件进行操作。运行一个文件可以有许多不同的操作方法，异步 I/O 操作使用一个称为 *struct aiocb* 的数据结构体进行控制，它在 *aio.h* 中定义为：

```
struct aiocb
{
    int aio_fildes;              /* 文件描述符。 */
    int aio_lio_opcode;          /* 需执行的操作。 */
```

```
        int aio_reqprio;                  /* 优先偏移量要求。 */
        volatile void *aio_buf;           /* 缓存位置。 */
        size_t aio_nbytes;                /* 改变长度。 */
        struct sigevent aio_sigevent;     /* 信号数值。 */

/* 内部元素 */
        struct aiocb *__next_prio;
        int __abs_prio;
        int __policy;
        int __error_code;
        __ssize_t __return_value;

#ifndef __USE_FILE_OFFSET64
        __off_t aio_offset;               /* 文件偏移量. */
        char __pad[sizeof (__off64_t) - sizeof (__off_t)];
#else
        __off64_t aio_offset;             /* 文件偏移量. */
#endif
        char __unused[32];
};
```

POSIX.1b 标准要求 struct aiocb 结构体中至少包含下列表中所述的元素。在执行操作中可能存在更多的元素，但是要看这些元件是否是非便携式和不鼓励使用的元素。

nt aio_fildes

该元素确定了用于操作的文件描述符。它必须是一个合法的描述符，否则将导致操作失败。

打开文件的设备必须允许搜索操作。即：在终端设备中不能使用任何 AIO 操作，否则 lseek 调用将会出现错误。

off_t aio_offset

此元素确定了执行操作（输入或输出）文件中的偏移量，因为操作是以任意命令进行，而且对于一个文件描述符而言，可用几种操作方式进行操作，因此，文件描述符没有确定的当前阅读/编写位置。

*volatile void *aio_buf*

这是一个缓存指示器，该缓存中有所需编写的数据或者有存储读取数据的空间。

Size_t aio_nbytes

此元素确定了 *aio_buf* 指定的缓存长度。

int aio_reqprio

若操作平台已经定义为 **_POSIX_PRIORITIZED_IO** 和

_POSIX_PRIORITY_SCHEDULING，则根据当前规定的优先权对 AIO 要求进行处理。可以用 *aio_reqprio* 元素降低 AIO 操作的优先权。

struct sigevent aio_sigevent

此元素确定了调用程序在操作终止时是如何得到通知的。若 *sigev_notify* 元素是 **SIGEV_NONE**，将不会发出通知信号；若是 **SIGEV_SIGNAL**，则发出 *sigev_signo* 确认的信号。否则 *sigev_notify* 必须是 **SIGEV_THREAD** 生成的信号，并执行 *sigev_notify_function* 指定的功能。

int aio_lio_opcode

此元素仅用于 *lio_listio* 和 *l io_listio64* 功能。因为这些功能允许一次进行多个任意操作，且每一个操作都可输入或输出（或不进行操作），信息必须存储于控制块中，可能的数值是：

LIO_READ

启动读取操作，读取在 *aio_offset* 位置的文件，并在内存中保存下一个 *aio_buf 指定的 aio_nbytes* 字节。

LIO_WRITE

启动编写操作，将 *aio_buf* 起始位置的 *aio_nbytes* 字节写入在 *aio_offset* 起始位置文件。

LIO_NOP

对于该控制块不做任何操作，有时在 *struct aiocb* 列数值存在缺陷时，该数值起作用。即，尽管在 *lio_listio* 中显示整各数列，也不必对一些数值进行处理。

当使用 32 字节机器上的 **_FILE_OFFSET_BITS == 64** 编辑数据源时，此类数据源事实上是 *struct aiocb64，*因为 LFS 界面代替了 *struct aiocb* 定义。

若使用 LFS 中定义的 AIO 功能，也存在着相似的定义类型，它们使用更大类型替换一定元素，但是与 struct aiocb 相同，且所有名称都相同。

```
/* 与 64bit 偏移量相同。请注意元素 aio_fildes
   to __return_value 在 aiocb 中和 aiocb64 需相同。  */
#ifdef __USE_LARGEFILE64
struct aiocb64
{
   int aio_fildes;                     /* 文件描述符。  */
   int aio_lio_opcode;                 /* 要完成的操作。  */
   int aio_reqprio;                    /* 要求优先权偏移量。  */
   volatile void *aio_buf;             /* 缓存位置。  */
   size_t aio_nbytes;                  /* 移动长度  */
   struct sigevent aio_sigevent;       /* 信号数值。  */

/* 内部元素  */
   结构体 aiocb *__next_prio;
```

```
    int __abs_prio;
    int __policy;
    int __error_code;
    __ssize_t __return_value;

    __off64_t aio_offset;              /* 文件偏移量。 */
    char __unused[32];
};
```

int aio_fildes

此元素确定了用于操作的文件描述符，该描述符必须是合法的描述，否则会出现由于明显原因而造成的操作失败。打开文件的设备必须允许搜索操作。即在终止设备上不能使用任何 AIO 操作，否则 lseek 调用将导致错误。

off64_t aio_offset

该元素确定了执行操作（输入或输出）的文件中的偏移量，因为该操作使用任意命令的方式进行，且使用了几种操作方法，因此不存在文件描述符的当前阅读/编写位置。

*volatile void *aio_buf*

这是一个缓存指示器，其中有要编写的数据或者存储数据的空间。

size_t aio_nbytes

此元素确定了 *aio_buf* 指定的缓存长度。

int aio_reqprio

若使用平台 **_POSIX_PRIORITIZED_IO** 且 **_POSIX_PRIORITY_SCHEDULING** 定义， AIO 要求根据当前规定的优先权进行处理，那么可用 *aio_reqprio* 降低 AIO 操作的优先权。

struct sigevent aio_sigevent

该元素确定了终止操作时调用程序如何得到通知。若信号是 *sigev_notify* ，元素是 **SIGEV_NONE**，则不发出通知；若是 **SIGEV_SIGNAL** 信号，则发出由 *sigev_signo* 确定的信号。否则，*sigev_notify* 必须是 **SIGEV_THREAD** 。在此情况下开始执行由 *sigev_notify_function* 指定的操作功能。

int aio_lio_opcode

该元素仅用于 *lio_listio* 和 *lio_listio64* 功能，因为这些功能允许一次可以进行多种任意操作，且每一种操作都能够进行输入或输出操作（或不操作）。因此，信息必须存储于控制块中。见有关 *struct aiocb* 的可能数值的描述。

当使用 32 位机器上的_FILE_OFFSET_BITS == 64 编辑数据源时，该类数据源将以 struct aiocb64 名称存在，因为 LFS 代替了旧的界面。

aio_cancel

名字

aio_cancel—取消异步 I/O 操作请求。

摘要

#include <errno.h>

#include <aio.h>

int aio_cancel (int fildes ,struct*aiocb* *aiocbp)

描述

当处理一个或多个异步请求时，在有些情况下取消一个选择的操作会有效。例如，如果编写数据不再准确且数据必须立即重写时，该作用特别明显。举例说明，设定一个应用程序，在文件中写入数据，此状况下新的引入数据须写入文件中并由排队的请求更新。执行 POSIX AIO 操作提供这种功能，但是此功能并不能够强制取消操作要求。这取决于执行的操作，确定是否能够取消操作。因此使用该功能只是一个提示。执行 libaio 操作并不能执行在 POSIX 库中的取消操作功能。

aio_cancel 功能可用于取消一个或多个待处理请求。若 aiocbp 参数是 NULL，则该功能将试图取消所有要处理文件描述符 fildes （即，其 aio_fildes 部分为 fildes ）的待处理请求。若 aiocbp 不是 NULL，则 aio_cancel 会试图取消 aiocbp 指定的特殊操作请求。

若操作要求被成功读取，则发出终止操作请求的常规通知。即，根据控制这个操作的 *struct sigevent* 主体，在不产生任何情况下，发出信号或启动一个线程操作。若不能取消操作请求，则在完成操作后，终止通常的操作方法，当成够地取消了一个操作请求时，作为此操作要求参数的 *aio_error* 的调用返回为 **ECANCELED**，*aio_return* 的调用将返回为*-1*。若操作请求没有取消且仍然运行，则错误状态为 **EINPROGRESS**。当数据源以*_FILE_OFFSET_BITS == 64* 编辑时,该功能事实上是 *aio_cancel64*，因为 LFS 界面明显代替了常规执行操作。

返回值

AIO_CANCELED

若存在未终止的操作请求和成功取消的操作请求。

AIO_NOTCANCELED

若存在一个或多个未能取消的余留操作要求，须使用 *aio_error* 找出哪一个要求，或许是多个要求（在 *aiocbp* 中为 *NULL*）未被成功地取消。

AIO_ALLDONE

若所有操作请求已经终止时，*aio_cancel* 被称为回复值是

缺陷

若在执行 *aio_cancel* 操作时出现错误，则功能回复为*-1* ，且设置 *errno* 为下列数值之一。

EBADF 文件描述符 *fildes* 无效。

ENOSYS 不执行 *aio_cancel* 操作。

aio_cancel 64

名字

aio_cancel64—取消异步 I/O 请求。

摘要

#include <errno.h>

#include <aio.h>

int aio_cancel64 (int fildes, struct aiocb64 *aiocbp)

描述

该操作功能与 *aio_cancel* 的功能相似，唯一区别就是该参数是一个与 *struct*aiocb64 类型变量有关的参数。

当使用 *_FILE_OFFSET_BITS==64* 对数据源进行编辑时，该功能在 *aio_cancel* 的操作下起作用，因此这显然代替了 32 位机器上的小文件界面。

返回值

见 aio_cancel(3)。

缺陷

见 aio_cancel(3)。

aio_error

名字

aio_error—获得 AIO 操作状态。

摘要

#include <errno.h>

#include <aio.h>

int aio_error (const struct aiocb *aiocbp)

描述

功能 *aio_error* 确定由 *aiocbp* 指向的、*strut aiocb* 变量描述的请求的错误状态。

当操作是异步完成时（当使用 *aio_read* 、 *lio_listio* 和 *lio_listio* 时，模式为 LIO_NOWAIT ），必须知道是否已经终止了一项具体请求。若已终止，要知道结果是什么。当使用*_FILE_OFFSET_BITS == 64* 进行数据源编辑时，该功能事实上是 *aio_error64* 功能，因为 LFS 界面显然取代了常规执行。

返回值

若操作请求还未终止，返回值总是 *EINPROGRESS* 。*aio_error* 返回值或是 *0*，若成功的完成了操作求或其返回的数值是存储于 *errno* 变量中的数值，若操作请求须使用 *read* , *write* , 或 *fsync*f 才能完成。

缺陷

ENOSYS　若未执行该操作，仍可返回

EINVAL　若 aiocbp 参数不是一个异步操作的参数，该异步操作的返回状态尚不明确。

aio_error64

名字

aio_error64—返回操作错误。

摘要

#include <errno.h>

#include <aio.h>

int aio_error64 (const struct aiocb64 *aiocbp)

描述

该功能与 *aio_error* 相似，唯一区别就是其参数是一个与 *struct aiocb64* 类型的变量有关的参数。

当使用_FILE_OFFSET_BITS == 64 编辑数据源时，该功能在名为 aio_error 的操作下起作用，因此显然代替了 32 位机器上的小文件界面。

返回值

若操作请求未终止，则返回值一直是 *EINPROGRESS* 。 *aio_error* 返回值也许是 *0*，若成功地完成了操作请求或其返回的数值是存储于 *errno* 变量中的数值，若操作请求须使用 *read* , *write* , 或 *fsync*f 才能完成。

缺陷

见 *aio_error(3).*

aio_fsync

名字

aio_fsync—用磁盘上的 aio_fsync 同步一个文件的全部内核状态。

摘要

#include <errno.h>

#include <aio.h>

int aio_fsync (int op, struct aiocb aiocbp)

描述

当进行异步操作时，有时候有必要达到一致的操作状态，对于 AIO 而言就意味着，需要了解是否执行了某一操作要求或一组操作要求，可以通过等待系统在操作终止时发出的通知得知此情况。但这在有时候则意味着浪费资源（主要在计算时间）。而 POSIX.1b 可以确定两种功能，在大多连惯性操作中期到帮助作用。

若*_POSIX_SYNCHRONIZED_IO* 符号定义为 *unistd.h*，则 *aio_fsync* 和 *aio_fsync64* 功能是唯一的可操作功能。

调用该功能迫使所有 I/O 操作在使用该操作功能时进行排队，此时，文件描述符 aiocbp->aio_fildes 的调用操作变成同步的 I/O 操作完成状态。aio_fsync 操作立即复位，但是，只有当此文件描述符上的所有操作请求都已终止且文件同步后，才会发出以 aiocbp->aio_sigevent 方式描述的通知。这意味着，对于同步操作请求后排队的十分相同的文件描述符不受影响。

若 op 是 O_DSYNC，当调用 fdatasync 时，产生同步操作，这时，op 应是 O_SYNC，当调用 fsync 时，产生同步操作。

只要不产生同步操作，由 aiocbp 指定的相关主体的 aio_error 的调用返回 EINPROGRESS 操作。若不能成功地进行同步操作，aio_error 返回 0 。否则返回值是 fsync 或 fdatasync 操作功能设定为 errno 的变量的值。在此情况下，对于写入该文件描述符的数据的一致性，不能进行任何假设。

返回值

若操作要求能够成功排队，该操作功能的返回值是 *0* 。否则返回值是*-1* 和 *errno*。

缺陷

EAGAIN　　由于暂时缺乏资源的原因，操作请求不能排队。

EBADF　　文件描述符 *aiocbp->aio_fildes* 无效或不能打开进行编辑。

EINVAL　　执行的操作不支持 I/O 同步或 *op* 参数是不同于 *O_DSYNC* 和 *O_SYNC* 的参数

ENOSYS　　该功能不可执行。

当使用 *_FILE_OFFSET_BITS == 64* 编辑数据源时，该操作功能事实上是 *aio_return64* 的操作功能，因为 LFS 界面显然取代了常规执行的操作。

aio_fsync64

名字

aio_fsync64—使用磁盘上的 aio_fsync64 同步一个文件的全部内核状态。

摘要

#include <errno.h>

#include <aio.h>

int aio_fsync64 (int op, struct aiocb64 *aiocbp)

描述

该功能与 *aio_fsync* 相似，唯一区别就是其参数是与 *struct aiocb64* 类的变量相关的参数。

当使用_FILE_OFFSET_BITS == 64 编辑数据源时，该功能在以 aio_fsync 为名称的操作下起作用，因此显然代替了 32 位机器上小文件的界面。

返回值

见 *aio_fsync.*

缺陷

见 aio_fsync

aio_init

名字

aio_init—如何优化 AIO 执行的操作。

摘要

#include <errno.h>

#include <aio.h>

void aio_init (const struct aioinit *init)

描述

POSIX 标准没有确定 AIO 操作功能如何执行，这些功能可能会由系统调用，但也有可能以用户级对这些功能的模拟。

在编辑的时候，可执行的操作是使用线程处理排队操作请求的用户级操作。当该执行操作请求对限制做出一些决定时，最好避免 GNU C 库中硬限制。因此，根据个人的使用要求，GNU C 库为改变 AIO 的操作执行提供了一种操作方法。

struct aioinit

此数据类型用于将配置或可调参数传递到执行的操作中。该程序须初始化结构元素，并使用 *aio_init* 操作功能将其传递到执行的操作中。

int aio_threads

该元素确定可能在任何时间使用的线程最大值。

int aio_num

该数值提供了一个同时排队的操作请求的最大数值的估计值。

int aio_locks

非使用功能。

int aio_usedba

非使用功能。

int aio_debug

非使用功能。

int aio_numusers

非使用功能。

int aio_reserved[2]

非使用功能。

必须在任何其他 AIO 操作功能之前调用该功能。调用是自愿完成的，因为它只帮助 AIO 更好地完成执行的操作。

在调用 *aio_init* 操作功能之前，*struct aioinit* 型的变量元素功能必须初始化。再将作为参数的与此相关的变量传递到 *aio_init* 中，其本身可能或不可能注意到该提示。

它是一个按照在 *Irix*6 中的 SGI 操作功能建议的扩展名

aio_read

名字

aio_read—开始执行异步读取操作。

摘要

#include <errno.h>

#include <aio.h>

int aio_read (struct aiocb *aiocbp)

描述

该功能启动异步读取操作，当操作排队或遇到操作错误时，该异步操作立即复位。

文件的第一个 *aiocbp->aio_nbytes* 字节，对于 *aiocbp->aio_fildes* 而言，是一个写入缓存的描述符，该描述符在文件 *aiocbp->aio_offset* 中从 *aiocbp->aio_buf* 位置开始。

若区分优先次序的 I/O 受平台支持，那么，在实际排队操作请求前，*aiocbp->aio_reqprio* 值用于调节其优先权。

根据 *aiocbp->aio_sigevent* 值，在读取操作请求终止时通知调用过程。

返回值

当 *aio_read* 返回时,如果在排队过程前不出现能够发现的错误，则返回值是零。如果发现此类早期的错误，操作功能返回值是*-1* 且设定为 *errno*。

若 *aio_read* 返回值是零,可以使用 *aio_error* 和 *aio_return*the 功能对当前操作请求状态进行查询。只要 *aio_error* 的操作返回值是 EINPROGRESS ，那么该操作尚未完成。若 *aio_error* 返回值是零，那么表明已成功地终止了操作。否则，数值将被解为错误解码。若终止了操作功能，使用 *aio_return* 返回值的调用，可以获得操作结果。返回值与应该返回的同等的 *read* 调用返回值相同。 使用 *_FILE_OFFSET_BITS == 64* 编辑数据源时，该功能实际上是 aio_read64 的操作功能，因为 the LFS 界面显然代替了正常操作。

缺陷

在出现早期错误的情况下：

EAGAIN　因为（暂时）超过了数据源的限制，操作请求未被排队。

ENOSYS　不执行 *aio_read* 操作功能。

EBADF　*aiocbp->aio_fildes* 描述符无效。在排队操作请求之前，不能识别出该情况，因此也会异步发出该错误信号。

EINVAL　*aiocbp->aio_offset* 或 *aiocbp->aio_reqpiro* 值无效。在排队操作请求之前，不能识别出该情况，因此也会异步发出该错误信号。

在正常返回操作的情况下，由 *aio_error* 操作返回的错误码可能是：

EBAD　*aiocbp->aio_fildes* 描述符无效。

ECANCELED　在完成操作前取消操作。

EINVAL　*aiocbp->aio_offset* 值无效。

aio_read64

名字

aio_read64—开始异步读取操作。

摘要

```
#include <errno.h>
#include <aio.h>
int aio_read64 (struct aiocb *aiocbp)
```

描述

该功能与 *aio_read* 的操作功能相同。唯一的区别就是在 32 位机器上的操作，文件描述符应该用大文件的模式打开。在内部，*aio_read64* 使用与 *seek64l* 相同的函数性，为阅读操作准确定位文件描述符，这与用于 *aio_read* 中的 *lseek* 函数性相反。

当使用 *_FILE_OFFSET_BITS == 64* 编辑数据源时,该功能在名为 *aio_read* 的操作中有效，因此明显代替了在 32 位机器上小文件的界面。

返回值

见 *aio_read.*

缺陷

见 *aio_read.*

aio_return

名字

aio_return—重新恢复 I/O 异步操作状态。

摘要

#include <errno.h>

#include <aio.h>

ssize_t aio_return (const struct aiocb *aiocbp)

描述

该功能可用于重新恢复操作返回状态，该操作由指向 aiocbp 的变量中描述的操作请求执行。由 *aio_error* 执行的操作是 EINPROGRESS，且返回功能未定义。

操作请求一旦完成，该功能就可用于重新恢复返回数值。以下调用操作可能会导致未定义的操作行为。当使用 **_FILE_OFFSET_BITS == 64** 编辑数据源时，该操作功能实际上是 *aio_return64* 功能，因为 LFS 界面显然代替了正常操作。

返回值

返回值本身是由 *read* , *write* , 或 *fsync* 调用操作返回的值。

缺陷

操作功能可返回。

ENOSYS　　若未执行该操作。

EINVAL　　若 *aiocbp* 参数不是指未知的

返回状态的异步操作。

aio_return64

名字

aio_return64—重新恢复 I/O 的异步操作状态。

摘要

#include <errno.h>

#include <aio.h>

int aio_return64 (const struct aiocb64 *aiocbp)

描述

该功能与 *aio_return* 操作功能相似，唯一区别就是其函数是与 *struct aiocb64* 类型相关的变量。

当使用 *_FILE_OFFSET_BITS == 64* 编辑数据源时，该功能在以 *aio_return* 为名的操作功能下有效。因此很明显它代替了 32 位机器上的小文件的界面。

返回值

见 *aio_return*。

缺陷

见 *aio_return*。

aio_suspend

名字

aio_suspend—暂停操作直到一个具体设定的一项或多项操作请求终止为止。

摘要

#include <errno.h>

#include <aio.h>

int aio_suspend (const struct aiocb *const list[], int nent, const struct timespec *timeout)

描述

同步操作的另一种方法是：暂停操作直到一个具体设定的一项或多项操作请求被终止为止。使用 *aio_**操作通知开始操作有关的终止情况可达到此目的，但在有些情况下这并不是理想的方法。在经常更新程序、以某种方式连接到服务器的客户中，此办法对于 robin 程序而言，并不是最好的方法，因为有些连接会很慢。另一方面，让 *aio_**通知调用者也不会是最好的方法，因为每当程序给客户准备数据时，都不会受到一个通知的干扰，在为当前客户提供服务之前，不会受理新客户。鉴于此，应采用 *aio_suspend* 操作功能。

当调用此功能时，调用线程暂停，直到至少完成排列 *list* 中 *nent* 元素指定的一个操作要求。在 *aio_suspend* 调用期间，如果已经完成所有操作请求，则立即返回操作功能。是否终止一项操作请求，这取决于对操作请求的错误状态与 *EINPROGRESS* 进行比较的结果。若 *list* 的一个元素为 *NULL*，则会忽略该项请求。

若未完成操作请求，则暂停调用程序。若 *timeout* 为 *NULL*，该调用程序要等到一项操作请求完

成之后才开始进行操作，若 *timeout* 不是 *NULL*，至少在 *timeout aio_suspend* 操作以错误状态返回时，该程序一直处于暂停状态。

当使用*_FILE_OFFSET_BITS == 64* 编辑数据源时，该操作事实上是 *aio_suspend64* 的操作功能，因为 LFS 界面显然代替了正常操作执行。

返回值

若已经终止了 *list* 操作中的一项或多项请求，则功能返回值是 *0*。否则操作功能返回值是*-1* 且设定 *errno*。

缺陷

EAGAIN　在 *timeout* 确定的时间，没有完成任何一项 *list* 中的操作请求。

EINTR　信号干扰了 *aio_suspend* 的操作。在发出终止一项操作请求的信号同时，AIO 在执行操作也会发出该信号。

ENOSYS　未完成 *aio_suspend* 功能。

aio_suspend64

名字

aio_suspend64—暂停操作直到一个具体设定的一项或多项操作请求终止为止。

摘要

#include <errno.h>

#include <aio.h>

int aio_suspend64 (const struct aiocb64 *const list[], int nent, const struct timespec *timeout)

描述

此功能与 *aio_suspend* 的操作请求相似，唯一区别就是其参数是与 *struct aiocb64* 类相关的变量。

当使用 *_FILE_OFFSET_BITS == 64* 编辑数据源时，该功能在以 *aio_suspend* 为名称的操作下起作用，因此，显然代替了 32 位机器上小文件的界面。

返回值

见 *aio_suspend*。

缺陷

见 *aio_suspend*。

aio_write

名字

aio_write—开始一个异步写入操作。

摘要

#include <errno.h>

#include <aio.h>

int aio_write (struct aiocb * aiocbp);

描述

该操作是开始一个异步写入操作。在操作请求排队后或其发生前遇到错误后，该功能调用立返回。

始于 aiocbp->aio_buf 的缓存中的第一个 aiocbp->aio_nbytes 字节，作为 aiocbp->aio_fildes 的一个描述符被写入文件中，该描述符始于文件中 aiocbp->aio_offset 的绝对位置。

若平台支持优化的 I/O ，则在操作请求真正排队之前，用 aiocbp->aio_reqprio 的值调节优先权。

按照 aiocbp->aio_sigevent 的值，读取操作请求终止时通知调用程序。

当 aio_write 返回时，若在排队操作程序之前没有发现错误，则返回值为零。若发现该类早期错误，则操作功能返回值为-1，且 errno 设定为下列数值之一。

EAGAIN　　由于（暂时）超过数据源的限定，因此未排列操作请求。

ENOSYS　　未执行 aio_write 操作功能。

EBADF aiocbp->aio_fildes 描述符无效。在排队操作请求之前，也许不能识别此类状况，因此也会异步发出此操作错误的信号。

EINVALaiocbp->aio_offset 或 aiocbp->aio_reqprio 值无效。 在排队操作请求之前，也许不能识别此类状况，因此也会异步发出此错误信号。

在 aio_write 的返回值为零的情况下，可使用 aio_error 和 aio_return 操作功能查询当前操作请求状态。 只要 aio_error 的返回值为 EINPROGRESS，该操作就没有完成。若 aio_error 操作的返回值为零，则该操作已被成功地终止，否则，该操作值则解释为错误解码。若终止该该功能，可使用 aio_return 调用获得操作结果，这时，read 调用操作应该返回。aio_error 返回操作时可能的错误解码是：

EBADF　　aiocbp->aio_fildes 描述符无效。

ECANCELED　　在操作结束之前，取消该操作。

EINVAL　　aiocbp->aio_offset 值无效。

当使用 _FILE_OFFSET_BITS == 64 编辑数据源时，该操作实际是 aio_write64 的操作功能，因为显然 LFS 界面代替了常规操作。

返回值

aio_write 的操作返回时，在操作程序排队前没有错误，则返回值为零，若发现了该早期错误，那么，操作功能返回值为*-1*，且 *errno* 设定为下列数值之一。

缺陷

EAGAIN　由于（暂时）超过了数据源的限定，因此不排队操作请求。

ENOSYS　不执行 *aio_write* 的操作功能。

EBADF　*aiocbp->aio_fildes* 的描述符无效。在排列操作请求之前，也许不能识别此类状况，因此也会异步发出此错误信号。

EINVAL　*aiocbp->aio_offset* 或 *aiocbp->aio_reqprio* 值无效。在排列操作请求之前，也许不能识别此类状况，因此也会异步发出此错误信号。

aio_write64

名字

aio_write64—启动异步写入操作。

摘要

#include <errno.h>

#include <aio.h>

int aio_write64 (struct aiocb *aiocbp)

描述

该操作功能与 *aio_write* 的操作功能相似。唯一区别就是在 32 位机上的文件描述符应以大文件的模式打开，在内部，*aio_write64* 使用与 *lseek64* 相同的函数性，为写入准确定位文件描述符，它与 *aio_write* 中使用的 *lseek* 函数性相反。

当使用*_FILE_OFFSET_BITS == 64* 编辑数据源时，该功能在*以 aio_write* 为名称的操作下起作用，因此显然代替了 32 位机上的小文件界面。

返回值

见 *aio_write*。

缺陷

见 *aio_write*。

io

名字

io—异步 IO 操作。

摘要

#include <errno.h>

#include <libio.h>

描述

libaio 库定义新 I/O 操作集，该操作集能够极大减少应用程序在 I/O 上的等待时间。新功能允许程序启动一个或多个 I/O 操作程序。然后，在并行执行 I/O 操作的同时，立即恢复正常工作。

这些功能是名为 *libaio libc* 二元实时功能库的一部分。这些功能使用内核支持而得以实现。

所有 IO 操作都在先前打开的文件上执行。一个文件可能需要多个任意操作。异步的 I/O 操作由名为 *struct iocb* 的数据结构体控制。它在 *libio.h* 中的定义如下：

```
typedef struct io_context *io_context_t;

typedef enum io_iocb_cmd {
        IO_CMD_PREAD = 0,
        IO_CMD_PWRITE = 1,

        IO_CMD_FSYNC = 2,
        IO_CMD_FDSYNC = 3,

        IO_CMD_POLL = 5,
        IO_CMD_NOOP = 6,
} io_iocb_cmd_t;

struct io_iocb_common {
        void            *buf;
        unsigned        __pad1;
        long            nbytes;
        unsigned        __pad2;
        long            longoffset;
        long long       __pad3, __pad4;
};      /* 结果代码是读取的数量或 -’ve errno */

struct iocb {
        void            *data;
        unsigned        key;
        short           aio_lio_opcode;
        short           aio_reqprio;
```

```
        int                     aio_fildes;
        union {
                struct io_iocb_common           c;
                struct io_iocb_vector           v;
                struct io_iocb_poll             poll;
                struct io_iocb_sockaddr saddr;
        } u;
};
```

int aio_fildes

该元素指定用于操作的文件描述符。它必须是合法描述符，否则该操作将失败。

打开文件的设备必须允许查找操作。也就是，在 *lseek* 调用会导致错误的终端设备上不可能使用任何 IO 操作。

long u.c.offset

该元素指定执行操作（输入或输出）的文件的偏移量。由于操作可以以任意顺序执行，并且一个文件描述符能够启动一个以上的操作，所以无法获得该文件描述符的当前读/写位置。

*void *buf*

这是缓存指针，指向要写的数据或读取数据的存储位置。

long u.c.nbytes

该元素指定由 *io_buf 指向的*缓存的长度。

int aio_reqprio

当前未使用。

IO_CMD_PREAD

启动读取操作。在 *u.c.offset* 位置上读取，将下 *u.c.nbytes* 个字节存储在 *buf 指向的缓存中。*

IO_CMD_PWRITE

启动写操作。把以 *buf* 开始的 *u.c.nbytes* 字节，写入以 *u.c.offset* 开始的文件中。

IO_CMD_NOP

不用改动此控制区。有时，当一批 *struct iocb* 值中包含缺陷时，该值很有用。也就是，尽管整批传递给 *io_submit* 功能，但是有些值不能被处理。

IO_CMD_FSYNC

IO_CMD_POLL

这是实验性的。

例子

```
/*
```

```
* 使用 async i/o 复制命令的简单化版本
*

* 来自:      Stephen Hemminger shemminger@osdl.org
    *通过使用 async I/O 状态机复制文件。
    *1. 启动读请求
    *2. 当读完成时，将其转为写请求
    * 3. 当写完成时，计数器减量，释放资源
    *
    *
    * 用法：aiocp 文件目的
    */

    #include <unistd.h>
    #include <stdio.h>
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <sys/param.h>
    #include <fcntl.h>
    #include <errno.h>

    #include <libaio.h>
#define AIO_BLKSIZE       (64*1024)
#define AIO_MAXIO                 32
static int busy = 0;                    // 运行中的 I/O #
static int tocopy = 0;                  // 剩余复制区 #
static int dstfd = -1;                  // 目地文件描述符

    static const char *dstname = NULL;
    static const char *srcname = NULL;

    /* 致命错误处理器 */
    static void io_error(const char *func, int rc)
    {
    if (rc == -ENOSYS)
```

```
fprintf(stderr, "AIO not in this kernel0);
        else if (rc < 0 && -rc < sys_nerr)
fprintf(stderr, "%s: %s0, func, sys_errlist[-rc]);
        else
fprintf(stderr, "%s: error %d0, func, rc);
        if (dstfd > 0)
close(dstfd);
        if (dstname)
unlink(dstname);
        exit(1);
        }

        /*
        * 写完成返回。
        * 调整记数，并释放资源
        */
        static void wr_done(io_context_t ctx, struct iocb *iocb, long res, long res2)
        {
        if (res2 != 0) {
io_error("aio write", res2);
        }
        if (res != iocb->u.c.nbytes) {
fprintf(stderr, "write missed bytes expect %d got %d0, iocb->u.c.nbytes, res2);
exit(1);
        }
        --tocopy;
        --busy;
        free(iocb->u.c.buf);
memset(iocb, 0xff, sizeof(iocb));        // 错乱
        free(iocb);
        write(2, "w", 1);
        }

        /*
        * 读完成返回。
        * 将读 iocb 更改为写 iocb，并启动。
```

```
*/
static void rd_done(io_context_t ctx, struct iocb *iocb, long res, long res2)
{
/* 库需要附件查看 iocb？ */
int iosize = iocb->u.c.nbytes;
char *buf = iocb->u.c.buf;
off_t offset = iocb->u.c.offset;

if (res2 != 0)
io_error("aio read", res2);
if (res != iosize) {
fprintf(stderr, "read missing bytes expect %d got %d0, iocb->u.c.nbytes, res);
exit(1);
}

/* 将读转为写 */
io_prep_pwrite(iocb, dstfd, buf, iosize, offset);
io_set_callback(iocb, wr_done);
if (1 != (res = io_submit(ctx, 1, &iocb)))
io_error("io_submit write", res);
write(2, "r", 1);
}

int main(int argc, char *const *argv)
{
int srcfd;
struct stat st;
off_t length = 0, offset = 0;
io_context_t myctx;
if (argc != 3 || argv[1][0] == '-') {
fprintf(stderr, "Usage: aiocp SOURCE DEST");
exit(1);
}
if ((srcfd = open(srcname = argv[1], O_RDONLY)) < 0) {
perror(srcname);
exit(1);
}
```

```
        if (fstat(srcfd, &st) < 0) {
    perror("fstat");
    exit(1);
        }
        length = st.st_size;
        if ((dstfd = open(dstname = argv[2], O_WRONLY | O_CREAT, 0666)) < 0) {
    close(srcfd);
    perror(dstname);
    exit(1);
        }

        /* 初始化状态机 */
        memset(&myctx, 0, sizeof(myctx));
        io_queue_init(AIO_MAXIO, &myctx);
        tocopy = howmany(length, AIO_BLKSIZE);

    while (tocopy > 0) {

int i, rc;

/* 立即提交尽可能多的读，最大可达 AIO_MAXIO */

int n = MIN(MIN(AIO_MAXIO - busy, AIO_MAXIO / 2),

                                howmany(length - offset, AIO_BLKSIZE));

if (n > 0) {

struct iocb *ioq[n];

for (i = 0; i < n; i++) {

                                struct iocb *io = (struct iocb *) malloc(sizeof(struct
                                iocb));

                                int iosize = MIN(length - offset, AIO_BLKSIZE);

                                char *buf = (char *) malloc(iosize);

                                if (NULL == buf || NULL == io) {

                                fprintf(stderr, "out of memory0);
```

```
                                                exit(1);
                                                }
                                                io_prep_pread(io, srcfd, buf, iosize, offset);
                                                io_set_callback(io, rd_done);
                                                ioq[i] = io;
                                                offset += iosize;
}
rc = io_submit(myctx, n, ioq);
if (rc < 0)
                                                io_error("io_submit", rc);
busy += n;
}
// 处理已完成 I/O
rc = io_queue_run(myctx);
if (rc < 0)
io_error("io_queue_run", rc);
// 如果运行中的 I/O 已达最大数量
// 那么等待一个完成
if (busy == AIO_MAXIO) {
rc = io_queue_wait(myctx, NULL);
if (rc < 0)
                                                io_error("io_queue_wait", rc);
}
        }
```

```
close(s rcfd);
close(dstfd);
exit(0);
}

/*
* 结果的特征为：
* [alanm@toolbox ~/MOT3]$ ../taio kernel-source-2.4.8-0.4g.ppc.rpm abc
* rrrrrrrrrrrrrrwwwrwrrwwrrwrwwrrwrwrwwrrwrwrrrrwwrwwwrrwrrrwwwwwwwwwwwwwwwwww
* rrrrrrrrrrrrrwwwrrwrwrwrwrrwwwwwwwwwwwwwwwrrrrrrrrrrrrrrrrrrwwwwrwrwwrwrwrwr
* wrrrrrrrwwwwwwwwwwwwwwwwrrrwrrrwrrwrwwwwwwwwwwwwwwrrrrwwrwrrrrrrrrrrrrwwwwwwwwwwwwwrww
* wwwrrrrrrrrrwwrrrwwrwrwrwwwrrrrrrrwwwrrwwwrrwrwwwwwwwwwwrrrrrrrwwwrrrrrrrwwwwww
* wwwwwwwrwrrrrrrrrwrrwrrwrrwrwrrrwrrrwrrrwrwwwwwwwwwwwwwwwwwwwwwwwrrrwwwrrrrrrrrrr
* rrwrrrrrrwrrwwwwwwwwwwwwwwwwwwwrwwwrrwrwwrrrrrrrrrrrrrrrrrrrrwwwwwwwwwwwwwwwwwwwww
* rrrrrwrrwrwrwrrwrrrwwwwwwwwwwrrrrwrrrwrwwrwrrrwrrwrrrrwwwwwwwwrwrwwwwrwwrrrwrrr
* rrrwwwwwwwwrrrrwwrrrrrrrrrrrrrrwrwrrrrwwwwwwwwwwwwwwwwwrwrrrrwwwwrwrrrrwrwwwrrrwww
* rwwrrrrrrrrwrrrrrrrrrrrrrwwwwrrrwwwwrwrrwwwwwwwwwwwwwwwwwwwwwwwwwwrrrrrrrwwwwwwwwrw
*/
```

io_cancel

名字

io_cancel—取消 io 操作请求。

摘要

#include <errno.h>

#include <libaio.h>

int io_cancel(io_context_t ctx, struct iocb *iocb)

```
struct iocb {
    void            *data; /* 返回 io 完成事件中*/
    unsigned        key;
                    /* 用于识别 io 请求 *
    short
                    aio_lio_opcode;
    short
                    aio_reqprio;          /* 未使用 */
```

```
int              aio_fildes;
                 };
```

描述

试图取消先前传递给 io_submit 的 iocb。如果成功取消该操作，那么事结果件就会复制到结果指向的内存中，而不会放入完成排队中。

当处理一个或多个异步请求时，在有些情况下取消一个选择的操作会有效。例如，如果编写数据不再准确且数据必须立即重写时，该作用特别明显。 举例说明，设定一个应用程序，在文件中写入数据，此状况下新的引入数据须写入文件中并由排队的要求更新。

返回值

成功时返回 0 ，否则返回错误号。

缺陷

EFAULT　　如果指向的任何数据结构无效。

EINVAL　　如果 ctx_id 指定的 aio_context 无效。

EAGAIN　　如果未取消指定的 iocb。

ENOSYS　　如果未执行。

io_fsync

名字

io_fsync—在磁盘上的 io_fsync 同步一个文件的全部内核状态。

摘要

```
#include <errno.h>
#include <libaio.h>
int io_fsync(io_context_t ctx, struct iocb *iocb, io_callback_t cb, int fd)
struct iocb {

    void            *data;
    unsigned        key;
    short           aio_lio_opcode;
    short           aio_reqprio;
    int             aio_fildes;
};
typedef void (*io_callback_t)(io_context_t ctx, struct iocb *iocb, long res, long res2);
```

描述

处理异步操作时，有时进入一致状态是必要的。对 AIO（异步 I/O POSIX 标准）而言，有必要知道是否处理了某个特定请求或某组请求。这只能在操作终止后，通过等待系统发出通知来完成，但有时这意味着浪费资源（主要是计算时间）。

调用该功能迫使所有 I/O 操作在使用该操作功能时进行排队，此时，文件描述符 *iocb->io_fildes* 的调用操作变成同步的 I/O 操作完成状态。*io_fsync* 操作立即复位，但是，只有当此文件描述符上的所有操作请求都已终止且文件同步后，才会发出以 *io_callback* 方式描述的通知。这意味着，对于同步操作请求后排队的十分相同的文件描述符不受影响。

返回值

返回 0，否则返回错误号。

缺陷

EFAULT	*iocbs* 可以引用程序可到达地址空间之外的数据。
EINVAL	*ctx* 引用未初始化的 aio 语句，由 iocbs 指向的 iocb 包含初始化不正确的 iocb。
EBADF	iocb 包含不存在的文件描述符。
EINVAL	iocb 中指定的文件不支持给定的 io 操作。

io_getevents

名字

io_getevents—读取来自 io 操作请求的结果事件。

摘要

```
#include <errno.h>
#include <libaio.h>
struct iocb {
        void            *data;
        unsigned        key;
        short           aio_lio_opcode;
        short           aio_reqprio;
        int             aio_fildes;
};

struct io_event {
unsigned PADDED(data, __pad1);
```

unsigned PADDED(obj, __pad2);

unsigned PADDED(res, __pad3);

unsigned PADDED(res2, __pad4);

};

int io_getevents(io_context_t *ctx*, **long** *nr*, **struct io_event** ***events*[], **struct timespec** ***timeout*);

描述

试图从由 ctx 指定的 aio_context 完全队列中读取 nr 事件。

返回值

如果没有可用的事件，且超出 when 指定的超时（这里 when == NULL 可以指定无限长超时），那么将可能返回 0。注意：when 指向的超时是相对的，如果没有 NULL 和操作阻塞，它将被更新。如果未执行，ENOSYS 使用将失败。

缺陷

EINVAL　　如果 ctx_id 无效，如果 min_nr 超出范围，如果 nr 超出范围，如果 when 超出范围。

EFAULT　　如果任何指定的内存无效。

io_prep_fsync

名字

io_prep_fsync—在磁盘上同步一个文件的全部内核状态。

摘要

```
#include <errno.h>
#include <libaio.h>
static inline void io_prep_fsync(struct iocb *iocb, int fd)
struct iocb {

    void        *data;
    unsigned    key;
    short       aio_lio_opcode;
    short       aio_reqprio;
    int         aio_fildes;
};
```

描述

这是为 FSYNC 请求建立 iocbv 的内联方便功能。

文件为此

iocb->aio_fildes = fd

是用命令建立的描述符。

iocb->aio_lio_opcode = IO_CMD_FSYNC:

io_prep_fsync() 功能将建立 IO_CMD_FSYNC 操作，异步促使与 iocb 参数引用的 iocb 结构体 argumentaio_fildes 成员文件描述符相关的、调用 io_submit() 时排队的所有 I/O 操作进入同步 I/O 完全状态。当同步请求启动或排队到文件或设备时（甚至当数据不能立即同步时），该动能调用将返回。

如同调用 fsync()，完成当前所有排队的 I/O 操作；这就是说，如同同步 I/O 文件完整性完成所定义的那样。如果 io_prep_fsync() 排队操作失败，那么，对于 fsync()而言，不能保证完成特殊的 I/O 操作。

如果 io_prep_fsync() 成功，这只是进入相关完全状态的、调用 io_submit() 时排队的 I/O。不能保证在同步方式下该文件描述符并发的 I/O 完整性。

此功能立即返回。为了计划该操作，必须调用 *io_submit* 功能。

使用相同 iocb 的同时异步操作可以产生未定义的结果。

io_prep_pread

名字

io_prep_pread—建立异步读取。

摘要

#include <errno.h>

#include <libaio.h>

inline void io_prep_pread(struct iocb *iocb, int fd, void *buf, size_t count, long long offset)

```
struct iocb {

    void        *data;
    unsigned    key;
    short       aio_lio_opcode;
    short       aio_reqprio;
    int         aio_fildes;
};
```

描述

io_prep_pread 是内联方便功能，它为推动异步读取操作的 iocb 初始 化而设计。

该文件的第一个

iocb->u.c.nbytes = count

字节

iocb->aio_fildes = fd

是描述符，写入缓存的下列位置，开始于

iocb->u.c.buf = buf

读取开始于绝对位置

ioc->u.c.offset = offset

在文件中。

此功能立即返回。为了计划该操作，必须调用 *io_submit* 功能。

使用相同 iocb 的同时异步操作可以产生未定义的结果。

io_prep_pwrite

名字

io_prep_pwrite—建立异步写入的 iocb。

摘要

```
#include <errno.h>
#include <libaio.h>
inline void io_prep_pwrite(struct iocb *iocb, int fd, void *buf, size_t count, long long offset)
struct iocb {

    void            *data;
    unsigned        key;
    short           aio_lio_opcode;
    short           aio_reqprio;
    int             aio_fildes;
};
```

描述

io_prep_write 是建立并行写入的方便功能。

该文件的第一个

iocb->u.c.nbytes = count

字节

iocb->aio_fildes = fd

是描述符，写入缓存的下列位置，开始于

iocb->u.c.buf = buf

写入开始于绝对位置

ioc->u.c.offset = offset

在文件中。

此功能立即返回。为了计划该操作，必须调用 *io_submit* 功能。

使用相同 iocb 的同时异步操作可以产生未定义的结果。

io_queue_release

名字

io_queue_release—释放与用户空间句柄相关的范围。

摘要

#include <errno.h>

#include <libaio.h>

int io_queue_release(io_context_t ctx)

描述

io_queue_release 破坏与用户空间句柄相关的范围。可能取消任何待处理的 AIO，并阻碍完成。

cts。

返回值

成功时，**io_queue_release** 返回 0，否则返回 -error，这里 error 是一个在错误部分中定义的 Exxx 值。

缺陷

EINVAL　　*ctx* 是未初始化的 aio 范围，*iocbs* 指向的 iocb 包含初始化不正确的 iocb。

ENOSYS　　未执行

io_queue_init

名字

io_queue_init—初始化异步 io 状态机。

摘要

#include <errno.h>

#include <libaio.h>

int io_queue_init(int maxevents, io_context_t 　*ctx)

描述

io_queue_init 尝试创建至少能够接收 *maxevents* 事件的 aio 范围。*ctx* 必须指向已有的 aio 范围，且必须在调用前初始化为 *0*。如果该操作成功， *cxtp 中填入结果柄。

返回值

成功时，**io_queue_init** 返回 0，否则返回 -error，这里 error 是一个在错误部分中定义的 Exxx 值。

缺陷

EFAULT 　*iocbs* 引用程序可用地址空间之外的数据。

EINVAL 　*maxevents* 为 <= 0 或 *ctx* 是无效内存位置。

ENOSYS 　未执行

EAGAIN 　*maxevents* > *max_aio_reqs* 这里 max_aio_reqs 是可调值。

io_queue_wait

名字

io_queue_wait— 等待 io 操作请求完成。

摘要

#include <errno.h>

#include <libaio.h>

int io_queue_wait(io_context_t ctx, const struct timespec *timeout)

描述

试图从为 ctx_id 指定的 aio_context 完全排队中读取事件。

返回值

如果没有可用事件，且超出 when 指定的超时（这里 when == NULL 可以指定无限长超时），那么将可能返回 0。注意：when 指向的超时是相对的，在不是 NULL 和操作阻碍时，它将被更新。如果未执行，-ENOSYS 返回失败。

成功时， **io_queue_wait** 返回 0，否则返回 -error，这里 error 是一个在错误部分中定义的

Exxx 值。

缺陷

EFAULT　iocbs 引用程序可用地址空间之外的数据。

EINVAL　*ctx* 是未初始化的 aio 语句，*iocbs* 指向的 iocb 包含初始化不正确的 iocb。

ENOSYS　未执行

io_set_callback

名字

io_set_callback—建立 io 全部回调功能。

摘要

#include <errno.h>

#include <libaio.h>

static inline void io_set_callback(struct iocb *iocb, io_callback_t cb)

```
struct iocb {

    void            *data;
    unsigned    key;
    short               aio_lio_opcode;
    short               aio_reqprio;
    int         aio_fildes;
};
typedef void (*io_callback_t)(io_context_t ctx, struct iocb *iocb, long res, long res2);
```

描述

如果调用程序使用来自 io_getevents 的原事件，不会完成回调。只有使用库助手才可以完成回调

io_queue_run

名字

io_queue_run—处理完成的 io 操作请求。

摘要

#include <errno.h>

#include <libaio.h>

int io_queue_run(io_context_t　ctx)

描述

io_queue_run 试图从为 ctx_id 指定的 aio_context 完全队列中读取所有事件。

返回值

如果无可用事件，返回 0。如果未执行， -ENOSYS 返回失败。

缺陷

EFAULT　iocbs 引用程序可用地址空间之外的数据。

EINVAL　*ctx* 是未初始化的 aio 范围，*iocbs* 指向的 iocb 包含初始化不正确的 iocb。

ENOSYS　未执行

io_submit

名字

io_submit—提交 io 请求。

摘要

#include <errno.h>

#include <libaio.h>

int io_submit(io_context_t *ctx***, long** *nr***, struct iocb ****iocbs***[]);**

```
struct iocb {

    void            *data;
    unsigned    key;
    short               aio_lio_opcode;
    short               aio_reqprio;
int         aio_fildes;
```

描述

io_submit 为给定的 io 范围 ctx 提交 *nr* iocbs 供其处理.

io_submit 功能能够同时排队任意数量的读取和写入请求。所有请求可以对应同一文件、不同文件或者两者之间的每一种解决方案。

io_submit 从由 *iocbs* *aio_lio_opcode* 成员指向的排列得到 *nr* 请求，该成员的每个 *iocbs* 元素字

段是 **IO_CMD_PREAD**，有一个读取操作排队，类似于此排列元素的 *io_prep_pread* 调用（除非信号终止的方式不同，下面我们将看到的）。如果 *aio_lio_opcode* 成员是 **IO_CMD_PWRITE**，有一个写入操作排队。否则，*aio_lio_opcode* 必须是 **IO_CMD_NOP**，在这种情况下，此 *iocbs* 元素会被忽略。在有固定 *struct iocb* 元素排队，且一次仅处理一些需求的情况下，此“操作”很有用。另一种情况就是在处理所有请求和重新发布所余请求前，取消 *io_submit* 调用。

iocbs 指向队列每一元素的其他成员，必须有与操作适合的值，如同上述 *io_prep_pread* 和 *io_prep_pwrite* 文档所描述那样。

所有请求排队后，功能立即返回。一旦成功，**io_submit** 返回成功提交的 iocbs 数量。否则，返回 -error。error 是一个在错误区定义的 Exxx 值。

如果检测到错误，那么行为不明确。

使用相同 iocb 的同时异步操作，可以产生不明确的结果。

缺陷

EFAULT　　iocbs 引用程序可用地址空间之外的数据。

EINVAL　　*ctx* 引用未初始化的 aio 范围，*iocbs* 指向的 iocb 包含初始化不正确的 iocb。

EBADF　　iocb 包含的文件描述符不存在。

EINVAL　　iocb 中指定的文件不支持给定的 io 操作。

lio_listio

名字

lio_listio—列表定向 I/O

摘要

#include <errno.h>

#include <libaio.h>

int lio_listio (int mode, struct aiocb *const list[], int nent, struct sigevent *sig)

描述

除过或多或少具有传统界面的这些功能外，POSIX.1b 也定义了一个能够同时启动多个操作的功能，此功能能够自由处理混合读取和写入操作。因而，它类似于 *readv* 和 *writev* 的结合。

lio_listio 功能能够同时排队任意数量的读区和写入请求。所有请求可以对应同一文件、不同文件或者两者之间的每一种解决方案。

lio_listio io_submit 从由 *list aio_lio_opcode* 成员指向的排队中得到 *nent* 请求，该成员的每个 *iocbs* 元素字段都是 **LIO_READIO_CMD_PREAD**，有一个读取操作排队。类似于此排队元素的 *aio_read* 调用（除非信号终止的方式不同，下面我们将看到的）。如果 *aio_lio_opcode* 成员是 **LIO_WRITE**，则有一个写入操作排队。否则，*aio_lio_opcode* 必须是 **LIO_NOP**，在这种情况下，此列表元素会被忽略。在有固定 *struct iocb* 元素排队，且一次仅处理其中一些的情况下，此“操作”很有用。另一种情况就是在处理所有请求和重新发布剩余请求前，取消 *lio_listio* 调用。

列表指向队列每一元素的其他成员，必须有与操作匹配的值，如同上述 *aio_read* 和 *aio_write* 文档所描述的那样。

所有请求排队后，*mode* 参数决定 *lio_listio* 的行为。如果 *mode* 是 **LIO_WAIT**，则它将等到所有请求终止。否则，*mode* 必须是 **LIO_NOWAIT**，在这种情形下，在所有请求排队后该功能立即返回。在这种情形下，根据 sig 参数，调用程序得到所有请求终止的通知。如果 sig 是 **NULL**，就不会发送通知。否则，发送一个信号或者开始一个线程，就像在 *aio_read* 或 *aio_write* 描述中提到的那样。

当源数据以 **_FILE_OFFSET_BITS == 64** 编辑时，此功能实际上是 *lio_listio64*，因为 LFS 界面明显代替了正常操作执行。

返回值

如果 *mode* 是 **LIO_WAIT**，当所有请求成功完成时，*lio_listio* 返回的值为 0，否则函数返回 1， 并相应设置 *errno*。如果要找出那项或者那些请求失败，必须对排列列表的所有元素使用 *aio_error* 功能。

万一 *mode* 是 **LIO_NOWAIT**，如果所有请求正确排队，功能返回 0。可使用如上所述的 *aio_error* 和 *aio_return*，查找当前请求的状态。如果在此种方式下 *lio_listio* 返回 -1，就相应设置了一个全局变量 *errno* 。如果一项请求没有终止，*aio_error* 的调用返回 **EINPROGRESS**，值是不同的。这项请求完成，返回错误值（或者 0），使用 *aio_return* 可以检索处理结果。

缺陷

可能的错误值有：

EAGAIN　排队所有请求所需的资源当时无效。必须检查 *list* 的每个元素的错误状态，决定哪个请求失败。

另一原因可能是系统 AIO 请求宽度超出限制。在 GNU 系统上执行，不会发生这种情况，因为没有任意限制存在。

EINVAL　*mode* 参数无效，或者 *nent* 大于 **AIO_LISTIO_MAX**

EIO　一个或多个请求的 I/O 操作失败。应该检查每项请求的错误状态，决定哪项请求终止。

ENOSYS　不支持 *lio_listio* 函数。

如果 mode 参数是 LIO_NOWAIT，且调用者取消一次请求，由 aio_error 返回的此请求的错误状态为 ECANCELED。

lio_listio64

名字

Lio_listio64—列表定向 I/O。

摘要

#include <errno.h>

#include <libaio.h>

int lio_listio64 (int mode, struct aiocb *const list[], int nent, struct sigevent *sig)

描述

该功能类似 *code{lio_listio* 功能。唯一区别就是，在 32 位机上，文件描述符应该以大文件模式打开。在内部，*lio_listio64* 使用的函数性与 *lseek64"* 等同，正确定位读取和写入的文件描述符，她与 *lio_listio 中使用的 lseek* 功能*相反.*

当数据源使用 **_FILE_OFFSET_BITS == 64** 编辑时，该功能在以能够以 *lio_listio* 的名称下起作用，因而明显地替换 32 位机上的小文件界面。

附 录

附录 A：常见问题

此处搜集了与 Red Flag Advanced Server 4.0 系统相关的一些问题，并给出它们的解决办法。

➢ **安装 Red Flag Advanced Server 4.0 的第二张光盘后，EVMS 无法启动。**

为了避免与 RAID 或 LVM 产生冲突，EVMS 不会在系统启动时自动启动，如果要使用 EVMS，需要使用如下命令：

/etc/rc.d/init.d/evms start

➢ **在 Red Flag Advanced Server 4.0 中安装 Oracle9iR2 的几个注意事项。**

1. 在使用 oracle9iR2 时，需要把 oracle 用户的 stack size 值设置为 8192。可以用 ulimit 命令查看其当前值。

```
# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
stack size              (kbytes, -s) unlimited
cpu time               (seconds, -t) unlimited
max user processes              (-u) 2047
virtual memory          (kbytes, -v) unlimited
```

设置方法是：修改 /etc/security/limits.conf 文件，在其中添加如下两行后重新登录。

```
*       soft    stack    8192
*       hard    stack    8192
```

2. 增加 share memory 的最大值。默认值是 32M。建议至少增大到 1G。

cat /proc/sys/kernel/shmmax

33554432

echo 1073741824 > /proc/sys/kernel/shmmax

可以在 /etc/sysctl.conf 中增加一行，使系统在启动时自动设置好这个值。

kernel.shmmax = 1073741824

3. 另外，在安装过程中，会发生一个错误，需要解决。

错误提示信息如下：

* "Error in invoking target install of makefile /opt/oracle/product/9.2.0/ctx/lib/ins_ctx.mk"

同时，下列错误信息将出现在 $ORACLE_HOME/install/make.log 文件中：

/lib/libdl.so.2: undefined reference to `_dl_addr@GLIBC_PRIVATE'

/lib/libdl.so.2: undefined reference to `_dl_open@GLIBC_PRIVATE'

/lib/libdl.so.2: undefined reference to `_dl_close@GLIBC_PRIVATE'

/lib/libdl.so.2: undefined reference to `_dl_sym@GLIBC_PRIVATE'

/lib/libdl.so.2: undefined reference to `_dl_vsym@GLIBC_PRIVATE'

这时，需要编辑文件 $ORACLE_HOME/ctx/lib/env_ctx.mk，找到 "INSO_LINK =" 行，在其中添加一个 "$(LDLIBFLAG)dl" 后保存它。

下面是要添加 "$(LDLIBFLAG)dl" 标记的一行的完整内容：

INSO_LINK = -L$(CTXLIB) $(LDLIBFLAG)m **$(LDLIBFLAG)dl** $(LDLIBFLAG)sc_ca (LDLIBFLAG)sc_fa $(LDLIBFLAG)sc_ex $(LDLIBFLAG)sc_da $(LDLIBFLAG)sc_ut $(LDLIBFLAG)sc_ch $(LDLIBFLAG)sc_fi $(LLIBCTXHX) $(LDLIBFLAG)c-Wl,-rpath,$(CTXHOME)lib $(CORELIBS) $(COMPEOBJS)

附录 B：术语表

account

在 Unix 系统中，指允许个人连接到系统的登录名称、个人目录、密码以及 shell 的组合。

alias

别名。在 shell 中为了能在执行命令时将某一字符串替换成另一个的一种机制。在提示符中键入 alias 可了解当前所定义的全部别名。

ARP

Address Resolution Protocol（地址解析协议）。该网际网络协议用于将网际网络地址动态地对应到局域网络的硬件地址上。

ATAPI

AT Attachment Packet Interface，AT 附件包装接口。最为人们所熟知的是 IDE；它提供了额外的指令来控制 CDROM 以及磁带装置。而具有延伸功能的 IDE 控制器通常被称为 EIDE（Enhanced IDE，加强型 IDE 控制器）。

batch

批处理。将工作按顺序送到处理器，处理器一个接一个执行直到最后一个完成并准备好接受另一组处理清单的一种处理模式。

boot

引导。即发生在按下计算机的电源开关，机器开始检测接口设备的状态，并把操作系统加载到内存中的整个过程。

bootdisk

引导盘。包含来自硬盘（有时也可从其本身）加载操作系统的必要程序代码的可开机软磁盘。

BSD

Berkeley Software Distribution（伯克利软件发行套件）。一套由美国伯克利大学信息相关科系所发展的 Unix 分支。

buffer

缓冲区。指内存中固定容量一个小区域，其中的内容可以加载区域模式文件，系统分区表，以及执行中的进程等等。所有缓冲区的连贯性都是由缓冲区内存来维护的。

buffer cache

缓冲区存取。这是操作系统核心中甚为重要的一部份，负责让所有的缓冲区保持在最新的状态，在必要时可以缩小内存空间，清除不需要的缓冲区。

CHAP

Challenge-Handshake Authentication Protocol（询问交互式身份验证协议）：ISP 验证其客户端所采用的通信协议。它与 PAP 的不同处在于：进行最初的判别后，每隔固定的时间周期它将会重新再验证一次。

client

客户端。是指能够短暂地连接到其它程序或计算机上并对其下达命令或要求信息的一个程序或一部计算机。它是 **服务器/客户端系统** 组件的一部分。

client/server system

服务器/客户端系统。由一个 **server**（服务器端）与一个或多个 **client**（客户端）所组成的系统架构或通信协议。

compilation

编译。指把人们读得懂的以某种程序语言（例如 C 语言）书写的程序源代码转换成机器可读的二进制文件的一种过程。

completion

自动补齐。只要系统内有能与之配合对象，shell 将自动把一个不完全的子字符串，延展扩大成一个已存在的文件名、用户名或其它种种的能力。

compression

压缩。这是一种在通信连接的传送过程中缩小文件或减少字符数目的方法。压缩程序通常包含有 compress，zip，gzip 及 bzip2。

console

控制台。也就是人们一般使用并称为终端的概念。它们是连接到一部巨型中央计算机的使用者操作的机器。对 PC 而言，实际的终端就是指键盘与屏幕。

cookies

由远程 web 服务器写入到本地硬盘的临时文件。它让服务器可以在使用者再次连上网站的时候可以知道其个人偏好。

DHCP

Dynamic Host Configuration Protocol（动态主机配置协议）。一种以局域网络机器为设计基础，能从 DHCP 服务器动态取得 IP 地址的通信协议。

DMA

Direct Memory Access（直接内存存取）。一种运用在 PC 架构上的技术，它允许接口设备可以从主存

储器存取或读写资料而无须通过 CPU 联系。

DNS

Domain Name System（网络域名系统）。用来负责分配名称/地址的机制。它可以将机器名称对应到 IP 地址。同样 DNS 也允许反向搜寻，也就是说可以从 IP 地址得知其机器名称。

DPMS

Display Power Management System（显示器电源管理系统）。用于所有现今生产的显示器以管理其电源使之能够延长使用年限的协议。

editor

编辑器。一般而言是指编辑文本文件所使用的程序（也就是文字编辑器）。最为人所熟知的 GNU/Linux 编辑器有 Emacs 以及 VIM。

email

电子邮件。是处于相同网络里的人们互相传送电子信息的一种方式。与定期邮件相同，email 需要收件人以及寄件人地址以便正确地传送信息。

environment variables

环境变量。可以直接通过 shell 查看环境变量。

ext2

「Extended 2 filesystem」的简称。是 GNU/Linux 原有的文件系统并且有任何 Unix 文件系统的特色：支持特殊文件（字符设备，符号链结...），文件的权限与所有权等等。

FAT

File Allocation Table（文件配置表）。使用于 DOS 以及 Windows 操作系统上的文件系统。

FDDI

Fiber Distributed Digital Interface（光纤分配式数字接口）。一种用于光纤通信的高速网络物理层。

FIFO

First In，First Out（先进先出）。一种内容项目被取出是依据其放入顺序的数据结构或硬件缓冲区。管道是 FIFO 概念在实践中最为普遍的一个例子。

Filesystem

文件系统。为使文件储存在实际介质（硬盘、磁盘）上时能够保持其资料的一致性所做的一种规划方式。

firewall

防火墙。在局域网络的拓扑中，负有与外界网络联系节点责任的机器或专属设备；同时也负有过滤或

控制某些通信端口的活动以及确定哪些特定接口能够予以存取等多重任务。

framebuffer

视频缓冲区。将显示卡上的 RAM 对应到机器内存地址空间的一种技术。它允许应用程序存取显示卡上的 RAM 而无须与之直接沟通。

FTP

File Transfer Protocol（文件传输协议）。这是用于机器间彼此传输文件的标准网际网络通信协议。

gateway

网关。用来连接两个 IP 网段之间的网络设备。

GIF

Graphics Interchange Format（图形交换格式）。一种广泛用于 web 的影像文件格式，GIF 影像资料可被压缩或者存入动态画面。

GNU

GNU's Not Unix 的缩写。GNU 计划由 Richard Stallman 发起于 80 年代初期，其目标是要发展出一套 free 的操作系统（"free" 代表 "自由" 而非免费）。

GPL

General Public License（通用公共许可证）。其理念与所有的商业软件授权大不相同：对于软件本身的复制、修改以及重新散布没有任何的限制，用户可以取得源代码，唯一的限制是将它散布给他人时，对方也将因相同的权利而获益。

GUI

Graphical User Interface（图形用户接口）。使用菜单，按钮，以及图标等等组成窗口外观的一种计算机操作界面。

host

主机，计算机的一种称呼。一般而言对连接到网络上的计算机时才会使用这个名词。

HTTP

HyperText Transfer Protocol（超文本传输协议）。此种通信协议让您得以连上缤纷多彩的网站并取回 HTML 文件或档案。

HTML

HyperText Markup Language（超文本标记语言）。这种语言可以用来书写 web 网页文件。

inode

在 Unix 类的文件系统中用来指向文件内容的进入点。每个 inode 皆可由这种独特的方式作为识别，

且同时包含着关于其所指向档案的相关信息，如存取时间、类型、文件大小。

Internet

网际网络。这是一个连接世界上众多计算机的巨大网络。

IP address

IP 地址。一组在 Internet 上用来确认计算机的由四组数字组成的地址表示法，IP 地址看起来像是 192.168.0.1 这种样子。而机器本身的地址有二种类型：静态或动态。静态 IP 地址不会变动；而动态 IP 地址则是指每次重新连上网络时，IP 地址都会有所不同。

IP masquerading

IP 伪装。当使用防火墙时隐藏计算机真实 IP 地址以防止为外界所窥知的一种方法。传统上任何越过防火墙而来的外界网络连结所取得的是防火墙的 IP 地址。

IRC

Internet Relay Chat（网际网络接力聊天室）。一种网络上用来实时交谈的标准。它允许建立一个频道（channel）进行私人秘密会谈，还可以传输文件。

ISA

Industry Standard Architecture（工业标准结构）。用于个人计算机上非常早期的总线规格，它正慢慢地被 PCI 总线所取代。

ISDN

Integrated Services Digital Network（综合服务数字网络）。一组允许以单一线缆或光纤传送声音、数字网络服务及影像的通信标准。

ISO

International Standards Organization（国际标准化组织）。

ISP

Internet Service Provider（网络服务提供者）。是指对其顾客提供网络存取而不论其介质是采用电话还是专用线路的公司。

kernel

核心。这是操作系统的关键所在。核心负责分配资源并区分各个使用者的进程。它处理着允许程序与计算机硬件直接沟通的所有动作，包含管理缓冲区快速存取等等。

LAN

Local Area Network（本地端局域网络）。一般而言是指当机器以相同实体线缆连接时所构成的网络系统。

LDP

Linux Documentation Project（Linux 文件计划）。一个维护 GNU/Linux 文件的非营利组织。其最著名的成果为各式各样的 HOWTO 文件，除此之外它也维护着 FAQ，甚至是一些书籍。

loopback

一个机器连接到其本身的虚拟网络接口，它允许执行中的程序不必去考虑两个网络实体事实上都位于相同机器的这种特殊状况。

manual page

参考手册。包含指令及其用法定义，可以 man 这个指令查阅的小型文件。

MBR

Master Boot Record（主引导记录）。指可引导硬盘的第一扇区所使用的名称。MBR 中包含用来将操作系统加载到内存或开机加载程序（例如 LILO）的执行码，以及该硬盘的分区表。

MIME

Multipurpose Internet Mail Extensions（多用途网际网络邮件延伸格式）。在电子邮件里，以型态/子型态（type/subtype）形式描述其包含文件内容的一段字符串。

MPEG

Moving Pictures Experts Group（运动图像专家组）。一个制订影音压缩标准的 ISO 委员会；同时 MPEG 也是他们的算法名称。

NCP

NetWare Core Protocol（NetWare 核心协议）。由 **Novell** 公司定义的用以存取 Novell NetWare 系统的文件及打印服务的通信协议。

newsgroups

新闻群组。能由新闻或 USENET 客户端程序加以存取以便让人阅读或写入信息到某新闻群组的特定主题讨论区或新闻区。

NFS

Network FileSystem（网络文件系统）。提供通过网络来共享文件的网络文件系统。

NIC

Network Interface Controller（网络接口控制器）。安装到计算机上并提供对网络实体连接所使用的转接器，如 Ethernet 网卡。

NIS

Network Information Service（网络信息服务），NIS 的目的在于分享跨越 NIS 网域的共有信息，该 NIS

网域涵盖了整个局域网、部分的局域网或是数个局域网。它能够输出密码数据库，服务数据库，以及群组信息等等。

PAP

Password Authentication Protocol（密码认证程序）。一种许多 ISP 用来认证客户端的协议，在这一设计中，客户端会送出一组未经编码的 ID 和密码给 server。

patch

补丁。包含有需发布的源代码的修订列表，目的是为了增加新功能，修改 bug 或按某些实际需要去修正。

path

指定文件或目录在文件系统中的位置。在 GNU/Linux 中有两种不同的路径：**相对路径**指的是文件或目录相对于当前目录的位置；**绝对路径**指的是文件或目录相对于根目录的位置。

open source

开放源代码。其理念在于一旦允许广大的程序设计师可以共同使用及修改原始程序代码，最终将会产生出对所有人而言最有用的产品。一些受欢迎的开放源码程序包括 Apache，sendmail 以及 GNU/Linux。

PAP

Password Authentication Protocol（密码认证程序）。一种许多 ISP 用来认证客户端的协议，在这一设计中，客户端会送出一组未经编码的 ID 和密码给服务器。

PCI

Peripheral Components Interconnect。由 Intel 制定的总线规格，现在已成为 PC 架构中的总线标准。它是 ISA 的继承者，而且提供了许多服务：装置、设定信息、IRQ 分享、总线控制及其它更多的功能。

PCMCIA

Personal Computer Memory Card International Association（个人计算机存储卡国际协会）通常被简称为“PC Card”，是便携式计算机外接口的标准，如：调制解调器，硬盘，存储卡，以太网卡等。

pipe

一种特别的 Unix 文件形式。一个程序将资料写入 pipe，而另一个程序由 pipe 读出资料直到结束。管道采用 FIFO（先进先出），因此资料被另一个程序读入直到顺序结束。

pixmap

“pixel map” 的缩写。是 bitmapped 影像的一种。

PNG

Portable Network Graphics（可移植网络图像文件）。该文件格式主要是给 web 使用，它被设计成无专利的，以取代具有专利权的 GIF，而且也有一些附加的功能。

PNP

Plug’N’Play（随插即用）。首先被用于 ISA 装置以便新增设定的信息，如今更广泛地用于所有装置以便回显设定参数。正如我们所知，所有的 PCI 装置都是即插即用的。

POP

Post Office Protocol（邮局协议）。这种常见的通信协议用于从 ISP 下载电子邮件。

PPP

Point to Point Protocol（点对点通信协议）。是一种通过序列信号线来传送资料的通信协议。通常被用于传送 IP 封包到网际网络，也可以和其它的通信协议一起使用，如 Novell 的 IPX 协议。

preprocessors

前置处理器。指示编译器取代在源代码中特定资料或程序片段，例如 C 的前置处理器为 #include，#define 等。

process

进程。在操作系统中，一个进程是伴随着一个程序的执行产生的。

prompt

提示符号。在 shell 中，它是在光标前的字符串。在其后输入字符命令。

Protocol

通信协议是指不同的机器经由网络通信的方式，不管是用软件或硬件，它们定义了数据传输时的格式。有许多的有名的通信协议，如 HTTP，FTP，TCP，和 UDP 等。

proxy

代理服务器。一台位于某一网络和网际网络间的机器，主要任务是加速多数被广泛使用的通信协议（如 HTTP、FTP）。它包含了一个预置的快速存取，可以降低重复资料被再次要求的成本。

quota

配额限制是限制使用者对于磁盘空间使用的一种方法。在某些文件系统上，管理者可以对各个使用者的目录做不同的大小限制。

RAID

Redundant Array of Independent Disks。始于伯克利大学资料系的一个计划，目的是让储存的资料分散于同一数组但不同的磁盘上。

RAM

Random Access Memory（随机存取内存）。是指计算机的主存储器 “Random” 也指内存的任何一部分都能被直接存取。

read-only mode

只读模式。表示不能写入文件，只能读取内容，当然也不能修改或删除文件。

read-write mode

读写模式。表示文件是可以被写入的，可以读取或修改文件内容，如果拥有这一权限，也可以删除文件。

root

root 是任何 Unix 系统上的超级使用者。Root 负责管理并维护整个 Unix 系统。

RFC

Request For Comments（计算机与通信技术文件）。RFC 是官方的 Internet 标准文件，由 IETF（Internet Engineering Task Force）所发行。他们描述所有使用或被要求使用的协议，如果想知道某一种通信协议是如何运作的，就可以去找对应的 RFC 文件来读。

RPM

Redhat Package Manager（红帽子软件包管理器）。一种为了产生软件套件而由 **Red Hat** 开发的软件包格式。它被用于许多 GNU/Linux 发行版本上，包括红旗 Linux。

run level

运行级别。是一项关于只允许某些被选定的进程存在的系统设定。在文件 /etc/inittab 中清楚地定义每个运行级别有那些进程是被允许的。

SCSI

Small Computers System Interface（小型计算机系统接口），一种高效且允许多种不同外设都能使用的总线规格。不同于 IDE，SCSI 总线的效能并不会受限于外围能接受指令的速度。只有高阶的机器才会在主板上内建 SCSI 总线，一般的 PC 用另外插卡的方式。

server

服务器。为程序或计算机提供功能或服务让客户端可以连接进来执行命令或是取得其所需的信息。

shadow passwords

影子密码。Unix 中的一种密码管理方式，系统中某个不是所有人都能读取的档案中存放着加过密的密码，是现在很常用的一种密码系统。它也提供了密码时间限制的功能。

shell

shell 是操作系统核心的基本接口，它提供命令行让使用者输入指令以便执行程序或系统命令。所有 shell 都有提供命令行的功能以便自动执行任务或是常用但复杂的任务。这些 shell 命令类似于 DOS 操作系统中的批处理文件，但是更为强大。常见的 shells 有 Bash，sh，和 tcsh. 等。

SMB

Server Message Block 是 Windows（9x/2000 或 NT）所使用的通信协议，用于通过网络共享文件或打印机。

SMTP

Simple Mail Transfer Protocol（简单邮件传输协议），是一种用来传送电子邮件的协议。邮件传送代理者如 sendmail 或 postfix 都使用 SMTP，他们有时也会被称为 SMTP 服务器。

socket

一种符合于任何网络连结的文件形态。

TCP

Transmission Control Protocol（传输控制协议）。这是所有使用 IP 来传送网络封包中最可靠的通信协议。TCP 加入了必要的检查，在 IP 中来确保封包被传送。和 UDP 相反，TCP 在连接模式下运行，即在交换信息前，两端的机器就要先建立连接。

telnet

开启一个连接到远程主机，telnet 是进行远程登录最常用的方式，也有更好更安全的方式，如 ssh。

URL

Uniform Resource Locator（统一资源定位器）。一种统一且特殊格式的字符串用以分辨在网络上的资源。这个资源可能是一个文件，一个服务器或是其它。

virtual desktops

虚拟桌面。在 X 窗口系统中，可以提供多个桌面。这一功能可以使您灵活安排工作窗口，避免让大量的程序都挤在同一桌面上。

WAN

Wide Area Network（广域网络）。

window manager

窗口管理器。一个负责图形环境“看起来的感觉”的程序。主要负责处理窗口的标题栏，框架，按钮，主菜单和一些快捷键方式。